Type Stability in Julia

Avoiding Performance Pathologies in JIT Compilation

ARTEM PELENITSYN, Northeastern University, USA
JULIA BELYAKOVA, Northeastern University, USA
BENJAMIN CHUNG, Northeastern University, USA
ROSS TATE, Cornell University, USA
JAN VITEK, Northeastern University, USA and Czech Technical University in Prague, Czech Republic

As a scientific programming language, Julia strives for performance but also provides high-level productivity features. To avoid performance pathologies, Julia users are expected to adhere to a coding discipline that enables so-called type stability. Informally, a function is type stable if the type of the output depends only on the types of the inputs, not their values. This paper provides a formal definition of type stability as well as a stronger property of type groundedness, shows that groundedness enables compiler optimizations, and proves the compiler correct. We also perform a corpus analysis to uncover how these type-related properties manifest in practice.

CCS Concepts: • **Software and its engineering → Just-in-time compilers**; *Semantics*.

Additional Key Words and Phrases: method dispatch, type inference, compilation, dynamic languages

ACM Reference Format:
Artem Pelenitsyn, Julia Belyakova, Benjamin Chung, Ross Tate, and Jan Vitek. 2021. Type Stability in Julia: Avoiding Performance Pathologies in JIT Compilation. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 150 (October 2021), 26 pages. https://doi.org/10.1145/3485527

1 INTRODUCTION

Performance is serious business for a scientific programming language. Success in that niche hinges on the availability of a rich ecosystem of high-performance mathematical and algorithm libraries. Julia is a relative newcomer in this space. Its approach to scientific computing is predicated on the bet that users can write efficient numerical code in Julia without needing to resort to C or Fortran. The design of the language is a balancing act between productivity features, such as multiple dispatch and garbage collection, and performance features, such as limited inheritance and restricted-by-default dynamic code loading. Julia has been designed to ensure that a relatively straightforward path exists from source code to machine code, and its performance results are encouraging. They show, on some simple benchmarks, speeds in between those of C and Java. In other words, a new dynamic language written by a small team of language engineers can compete with mature, statically typed languages and their highly tuned compilers.

Writing efficient Julia code is best viewed as a dialogue between the programmer and the compiler. From its earliest days, Julia exposed the compiler's intermediate representation to users,

Authors' addresses: Artem Pelenitsyn, Northeastern University, USA, pelenitsyn.a@northeastern.edu; Julia Belyakova, Northeastern University, USA; Benjamin Chung, Northeastern University, USA; Ross Tate, Cornell University, USA; Jan Vitek, Northeastern University, USA and Czech Technical University in Prague, Czech Republic.

2475-1421/2021/10-ART150
https://doi.org/10.1145/3485527

encouraging them to (1) observe if and how the compiler is able to optimize their code, and (2) adapt their coding style to warrant optimizations. This came with a simple execution model: each time a function is called with a different tuple of concrete argument types, a new *specialization* is generated by Julia's just-in-time compiler. That specialization leverages the run-time type information about the arguments, to apply *unboxing* and *devirtualization* transformations to the code. The former lets the compiler manipulate values without indirection and stack allocate them; the latter sidesteps the powerful but costly multiple-dispatch mechanism and enables inlining of callees.

One key to performance in Julia stems from the compiler's success in determining the *concrete* return type of any function call it encounters. The intuition is that in such cases, the compiler is able to propagate precise type information as it traverses the code, which, in turn, is crucial for unboxing and devirtualization. More precisely, Julia makes an important distinction between concrete and abstract types: a concrete type such as `Int64`, is final in the sense that the compiler knows the size and exact layout of values of that type; for values of abstract types such as `Number`, the compiler has no information. The property we alluded to is called *type stability*. Its informal definition states that a method is type stable if the concrete type of its output is entirely determined by the concrete types of its arguments.[1] Folklore suggests that one should strive to write type-stable methods outright, or, if performance is an issue, refactor methods so that they become type stable.

Our goal in this paper is three-fold. First, we give a precise definition of type stability that allows for formal reasoning. Second, we formalize the relationship between type-stable code and the ability of a compiler to perform type-based optimizations; to this end, we find that a stronger property, which we call *type groundedness*, describes optimizable code more precisely. Finally, we analyze prevalence of type stability in open-source Julia code and identify patterns of type-stable code. We make the following contributions:

- An overview of type stability and its role in Julia, as well as the intuition behind stability and groundedness (Sec. 3).
- An abstract machine called Jules, which models Julia's intermediate representation, dynamic semantics, and just-in-time (JIT) compilation (Sec. 4).
- Formal definitions of groundedness and stability, and a proof that type-grounded methods are fully devirtualized by the JIT compiler (Sec. 4.3). Additionally, we prove that the JIT compilation is correct with respect to the dynamic-dispatch semantics (Sec. 4.4). Detailed proofs are available in the extended version of the paper [Pelenitsyn et al. 2021b].
- A corpus analysis of packages to measure stability and groundedness in the wild, find patterns of type-stable code, and identify properties that correlate with unstable code (Sec. 5).

The paper is accompanied by the artifact [Pelenitsyn et al. 2021a] reproducing results of Sec. 5.

2 JULIA IN A NUTSHELL

The Julia language is designed around multiple dispatch [Bezanson et al. 2017]. Programs consist of *functions* that are implemented by multiple *methods* of the same name; each method is distinguished by a distinct type signature, and all methods are stored in a so-called method table. At run time, the Julia implementation dispatches a function call to the *most specific* method by comparing the types of the arguments to the types of the parameters of all methods of that function. As an example of a function and its constituent methods, consider `+`; as of version 1.5.4 of the language, there are 190 implementations of `+`, each covering a specific case determined by its type signature. Fig. 1 displays custom implementations for 16-bit floating point numbers, missing values, big-floats/big-integers, and complex arithmetic. Although at the source-code level, multiple methods look similar

[1] https://docs.julialang.org/en/v1/manual/faq/#man-type-stability

```
# 184 methods for generic function "+":
[1] +(a::Float16, b::Float16) in Base at float.jl:398
[2] +(::Missing, ::Missing) in Base at missing.jl:114
[3] +(::Missing) in Base at missing.jl:100
[4] +(::Missing, ::Number) in Base at missing.jl:115
[5] +(a::BigFloat, b::BigFloat, c::BigFloat, d::BigFloat) in Base.MPFR at mpfr.jl:541
[6] +(a::BigFloat, b::BigFloat, c::BigFloat) in Base.MPFR at mpfr.jl:535
[7] +(x::BigFloat, c::BigInt) in Base.MPFR at mpfr.jl:394
[8] +(x::BigFloat, y::BigFloat) in Base.MPFR at mpfr.jl:363
...
```

Fig. 1. Methods from the standard library

to overloading known from languages like C++ and Java, the key difference is that those languages resolve overloading statically whereas Julia does that dynamically using multiple dispatch.

Julia supports a rich type language for defining method signatures. Base types consist of either bits types—types that have a direct binary representation, like integers—or structs. Both bits types and struct types, referred to as *concrete types*, can have supertypes, but all supertypes are *abstract types*. Abstract types can be arranged into a single-subtyping hierarchy rooted at `Any`, and no abstract type can be instantiated. The type language allows for further composition of these base types using unions, tuples, and bounded existential constructors; the result of composition can be abstract or concrete. Zappa Nardelli et al. [2018] gives a detailed discussion of the type language and of subtyping.

Any function call in a program, such as `x+y`, requires choosing one of the methods of the target function. *Method dispatch* is a multi-step process. First, the implementation obtains the concrete types of arguments. Second, it retrieves applicable methods by checking for subtyping between argument types and type annotations of the methods. Next, it sorts these methods into subtype order. Finally, the call is dispatched to the most specific method—a method such that no other applicable method is its strict subtype. If no such method exists, an error is produced. As an example, consider the above definition of `+`: a call with two `BigFloat`'s dispatches to definition 8 from Fig. 1.

Function calls are pervasive in Julia, and their efficiency is crucial for performance. However, the many complex type-level operations involved in dispatch make the process slow. Moreover, the language implementation, as of this writing, does not perform inline caching [Deutsch and Schiffman 1984], meaning that dispatch results are not cached across calls. To attain acceptable performance, the compiler attempts to remove as many dispatch operations as it can. This optimization leverages run-time type information whenever a method is compiled, i.e., when it is called for the first time with a novel set of argument types. These types are used by the compiler to infer types in the method body. Then, this type information frequently allows the compiler to devirtualize and inline the function calls within a method [Aigner and Hölzle 1996], thus improving performance. However, this optimization is not always possible: if type inference cannot produce a sufficiently specific type, then the call cannot be devirtualized. Consider the prior example of `x+y`: if `y` is known to be one of `BigFloat` or `BigInt`, the method to call cannot be determined. This problem arises for various reasons, for example, accessing a struct field of an abstract type, or the type inferencer losing precision due to a branching statement. A more detailed description of the compilation strategy and its performance is given in [Bezanson et al. 2018].

3 TYPE STABILITY: A KEY TO PERFORMANCE?

Designing performant Julia code is best understood as a conversation between the language designers, compiler writers, and the developers of widely used packages. The designers engineered Julia so that it is possible to achieve performance with a moderately optimizing just-in-time (JIT) compiler and with disciplined use of the abstractions included in the language [Bezanson et al. 2018]. For example, limiting the reach of `eval` by default [Belyakova et al. 2020] allows for a simpler compiler implementation, and constraining struct types to be final enables unboxing. The compiler's work is split between high-level optimizations, mainly devirtualization and unboxing, and low-level optimizations that are off-loaded to LLVM. This split is necessary as LLVM on its own does not have enough information to optimize multiple dispatch. Removing dispatch is the key to performance, but to perform the optimization, the compiler needs precise type information. Thus, while developers are encouraged to write generic code, the code also needs to be conducive to type inference and type-based optimizations. In this section, we give an overview of the appropriate coding discipline, and explain how it enables optimizations.

Performance. To illustrate performance implications of careless coding practices, consider Fig. 2, which displays a method for one of the Julia microbenchmarks, `pisum`. For the purposes of this example, we have added an identity function `id` which was initially implemented to return its argument in both branches, as well-behaved identities do. Then, the `id` method was changed to return a string in the impossible branch (`rand()` returns a value from 0 to 1). The impact of that change was about a 40% increase in the execution time of the benchmark (Julia 1.5.4).

```
function id(x)
  (rand() == 4.2) ? "x" : x
end


function pisum()
  sum = 0.0
  for j = 1:500
    sum = 0.0
    for k = 1:10000
      sum += id(1.0/(k*id(k)))
    end
  end
  sum
end
```

(a) Microbenchmark source code, redacted

```
julia> @code_warntype id(5)
Variables
  #self#::Core.Compiler.Const(id, false)
  x::Int64
Body::Union{Int64, String}
1 - %1 = Main.rand()::Float64
|   %2 = (%1 == 4.2)::Bool
+--       goto #3 if not %2
2 -       return "x"
3 -       return x

julia> @code_warntype pisum()
...
|   %20 = k::Int64
|   %21 = Main.id(k)::Union{Int64, String}
```

(b) Julia session

Fig. 2. A Julia microbenchmark (a) illustrating performance implications of careless coding practices: changing `id` function to return values of different types leads to longer execution because of the `Union` type of `id(..)`, which propagates to `pisum` (b).

When a performance regression occurs, it is common for developers to study the intermediate representation produced by the compiler. To facilitate this, the language provides a macro, `code_warntype`, that shows the code along with the inferred types for a given function invocation. Fig. 2 demonstrates the result of calling `code_warntype` on `id(5)`. Types that are imprecise, i.e., not concrete, show up in red: they indicate that concrete type of a value may vary from run to run. Here, we see that when called with an integer, `id` may return either an `Int64` or a `String`. Moreover, the imprecise return type of `id` propagates to the caller, as can be seen by inspecting

`pisum` with `code_warntype`. Such type imprecision can impact performance in two ways. First, the `sum` variable has to be boxed, adding a level of indirection to any operation performed therein. Second, it is harder for the compiler to devirtualize and inline consecutive calls, thus requiring dynamic dispatch.

Type Stability. Julia's compilation model is designed to accommodate source programs with flexible types, yet to make such programs efficient. The compiler, by default, creates an *instance* of each source method for each distinct tuple of argument types. Thus, even if the programmer does not provide any type annotations, like in the `id` example, the compiler will create method instances for *concrete* input types seen during an execution. For example, since in `pisum`, function `id` is called both with a `Float64` and an `Int64`, the method table will hold two method instances in addition to the original, user-defined method. Because method instances have more precise argument types, the compiler can leverage them to produce more efficient code and infer more precise return types.

In Julia parlance, a method is called *type stable* if its inferred return type depends solely on the types of its arguments; in the example, `id` is not type stable, as its return type may change depending on the input value (in principle). The definition of type stability, though, is somewhat slippery. The community has multiple, subtly different, informal definitions that capture the same broad idea, but describe varying properties. The canonical definition from the Julia documentation describes type stability as

> *"[...] the type of the output is predictable from the types of the inputs. In particular, it means that the type of the output cannot vary depending on the values of the inputs."*

However, elsewhere, the documentation also states that *"An analogous type-stability problem exists for variables used repeatedly within a function:"*

```
function foo()
    x = 1
    for i = 1:10
        x /= rand()
    end
    x
end
```

This function will always return a `Float64`, which is the type of `x` at the end of the `foo` definition, regardless of the (nonexistent) inputs. However, the manual says that it is a type stability issue nonetheless. This is because the variable `x` is initialized to an `Int64` but then assigned a `Float64` in the loop. Some versions of the compiler boxed `x` as it could hold two different types; of course, in this example, one level of loop unrolling would alleviate the performance issue, but in general, imprecise types limit compiler optimizations. Conveniently, the `code_warntype` macro mentioned above will highlight imprecise types for *all* intermediate variables. Furthermore, the documentation states that

> *"[t]his serves as a warning of potential type instability."*

Effectively, there are two competing, type-related properties of function bodies. To address this confusion, we define and refer to them using two distinct terms:

- *type stability* is when a function's return type depends only on its argument types, and
- *type groundedness* is when every variable's type depends only on the argument types.

Although type stability is strictly weaker than type groundedness, for the purposes of this paper, we are interested in both properties. The latter, type groundedness, is useful for performance of the function itself, as it implies that unboxing, devirtualization, and inlining can occur. The former, type stability, allows the function to be used efficiently by other functions: namely, type-grounded

functions may call a function that is only type stable but not grounded. For brevity, when the context is clear, we will refer to type stability and type groundedness as stability and groundedness in what follows.

Flavors of stability. Type stability is an inter-procedural property, and in the worst case, it can depend on the whole program. Consider the functions of Fig. 3. Function `f0` is trivially type unstable, regardless of the type of its input: if `good(x)` returns true, `f0` returns an `Int64` value, otherwise `f0` returns a `String`. Function `f1` is trivially type stable as all control-flow paths return a constant of `Int64` type. Function `f2` is type stable as long as the negation operator is type stable and returns a value of the same type as its argument. As an example, we show a method `-(::Float64)` that causes `f2(::Float64)` to lose type stability. This is a common bug where the function `(-)` either returns a `Float64` or `Int64` due to the constant `0` being of type `Int64`. The proper, Julia-style implementation for this method is to replace the constant `0` with `zero(x)`, which returns the zero value for the type of `x`, in this case `0.0`.

```
function f0(x)
  if (good(x))
    0
  else
    "not 0"
  end
end
```

```
function f1(x)
  if (good(x))
    0
  else
    1
  end
end
```

```
function f2(x)
  if (good(x))
    x
  else
    -x
  end
end
```

```
function -(x::Float64)
  if (x==0)
    0
  else
    Base.neg_float(x)
  end
end
```

Fig. 3. Flavors of stability: `f0` is unstable, `f1` is stable; `f2` is stable if `(-)` is stable; `(-)` is unstable.

The example of function `f2` illustrates the fact that stability is a whole-program property. Adding a method may cause some, seemingly unrelated, method to lose type stability.

Stability versus groundedness. Type stability of a method is important for the groundedness of its callers. Consider the function `h(x::Int64)=g(x)+1`. If `g(x)=x`, it follows that `h` is both stable and grounded, as `g` will always return an `Int64`, and so will `h`. However, if we define `g(x) = (x==2) ? "Ho" : 4`, then `h` suddenly loses both properties. To recover stability and groundedness of `h`, it is necessary to make `g` type stable, yet it does not have to be grounded. For example, despite the presence of the imprecise variable `y`, the following definition makes `h` grounded: `g(x) = let y =(x==2 ?"Ho" : 4) in x end`.

In practice, type stability is sought after when making architectural decisions. Idiomatic Julia functions are small and have no internal type-directed branching; instead, branches on types are replaced with multiple dispatch. Once type ambiguity is lifted into dispatch, small functions with specific type arguments are relatively easy to make type stable. In turn, this architecture allows for effective devirtualization in a caller, as in many cases, the inferred type at a call site will determine its callee at compilation time.

Thus, writing type-stable functions is a good practice, for it provides callers of those functions with an opportunity to be efficient. However, stability of callees is not a sufficient condition for the efficiency of their callers: the callers themselves need to strive for type groundedness, which requires eliminating type imprecision from control flow.

Patterns of instability. There are several code patterns that are inherently type unstable. For one, accessing abstract fields of a structure is an unstable operation: the concrete type of the field value depends on the struct value, not just struct type. In Julia, it is recommended to avoid abstractly

typed fields if performance is important, but they are a useful tool for interacting with external data sources and representing heterogenous data.

Another example is sum types (algebraic data types or open unions), which can be modeled with subtyping in Julia. Take a hierarchy of an abstract type `Expr` and its two concrete subtypes, `Lit` and `BinOp`. Such a hierarchy is convenient, because it allows for an `Expr` evaluator written with multiple dispatch:

```
run(e :: Lit)   = ...                    run(e :: BinOp) = ...
```

If we want the evaluator to be called with the result of a function `parse(s ::String)`, the latter cannot be type stable: `parse` will return values of different concrete types, `Lit` and `BinOp`, depending on the input string. If one does want `parse` to be stable, they need to always return the same concrete type, e.g. an S-expression-style struct. Then, `run` has to be written without multiple dispatch, as a big if-expression, which may be undesirable.

4 JULES: FORMALIZING STABILITY AND GROUNDEDNESS

To simplify reasoning about type stability and groundedness, we first define Jules, an abstract machine that provides an idealized version of Julia's intermediate representation (IR) and compilation model. Jules captures the just-in-time (JIT) compilation process that (1) specializes methods for concrete argument types as the program executes, and (2) replaces dynamically dispatched calls with direct method invocations when type inference is able to get precise information about the argument types. It is the type inference algorithm that directly affects type stability and groundedness of code, and thus the ability of the JIT compiler to optimize it. While Julia's actual type inference algorithm is quite complex, its implementation is not relevant for understanding our properties of interest; thus, Jules abstracts over type inference and uses it as a black box.

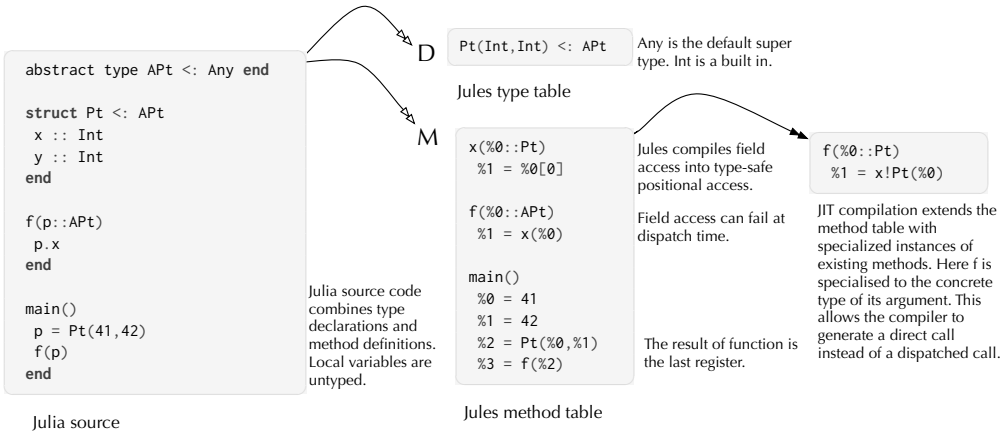

Fig. 4. Compilation from Julia to Jules

Fig. 4 illustrates the relationship between Julia source code, the Jules intermediate representation, and the result of compilation. We do not model the translation from the source to Jules, and simply assume that the front-end generates well-formed Jules code.[2] A Jules program consists of

[2]The front-end does not devirtualize function calls, as Julia programmers do not have the ability to write direct method invocations in the source.

an immutable *type table* D and a *method table* M; the method table can be incrementally extended with method instances that are compiled by the just-in-time compiler.

The source program of Fig. 4 defines two types, the concrete `Pt` and its parent, the abstract type `APt`, as well as two methods, `f` and `main`. When translated to Jules, `Pt` is added to the type table along with its supertype `APt`. Similarly, the methods `main` and `f` are added to the Jules method table, along with accessors for the fields of `Pt`, with bodies translated to the Jules intermediate representation.

The Jules IR is similar to static single assignment form. Each statement can access values in registers, and saves its result into a new, consecutively numbered, register. Statements can perform a dispatched call `f(%2)`, direct call `x!Pt(%0)`, conditional assignment (not shown), and a number of other operations. The IR is untyped, but the translation from Julia is type sound. In particular, type soundness guarantees that only dispatch errors can occur at run time. For example, compilation will produce only well-formed field accesses such as the one in `x(%0::Pt)`, but a dispatched call `x(%0)` in `f` could fail if `f` was called with a struct that did not have an `x` field. In order to perform this translation, Jules uses type inference to determine the types of the program's registers. We abstract over this type inference mechanism and only specify that it produces sound (with respect to our dynamic semantics) results.

Execution in Jules occurs between *configurations* consisting of both a stack of *frames* F (representing the current execution state) and a *method table* M (consisting of original methods and specialized instances), denoted F, M. A configuration F, M evolves to a new configuration F′, M′ by stepping F, M $\longrightarrow$ F′, M′; every step processes the top instruction in F and possibly compiles new method instances into M′. Notably, due to the so-called world-age mechanism [Belyakova et al. 2020] which restricts the effect of `eval`, source methods are fixed from the perspective of compilation; only compiled instances changes.

4.1 Syntax

The syntax of Jules methods is defined in Fig. 5. We use two key notational devices. First, sequences are denoted $\overline{\cdot}$; thus, $\overline{\mathsf{ty}}$ stands for types $\mathsf{ty}_0 \ldots \mathsf{ty}_n$, $\overline{\%\mathsf{k}}$ for registers $\%\mathsf{k}_0 \ldots \%\mathsf{k}_n$, and $\overline{\mathsf{st}}$ for instructions $\mathsf{st}_0 \ldots \mathsf{st}_n$. An empty sequence is written ϵ. Second, indexing is denoted $[\cdot]$; $\overline{\mathsf{ty}}[\mathsf{k}]$ is the k-th type in $\overline{\mathsf{ty}}$ (starting from 0), $\%\mathsf{j}[\mathsf{k}]$ is the k-th field of $\%\mathsf{j}$, and $\mathsf{M}[\mathsf{m}!\overline{\mathsf{ty}}]$ indexes M by method signature where m denotes method name and $\overline{\mathsf{ty}}$ denotes argument types.

ty	::=		type	st	::=		instruction
	|	T	*concrete type*		|	$\%\mathsf{i} \leftarrow \mathsf{p}$	*int. assignment*
	|	A	*abstract type*		|	$\%\mathsf{i} \leftarrow \%\mathsf{j}$	*reg. transfer*
					|	$\%\mathsf{i} \leftarrow \mathsf{T}(\overline{\%\mathsf{k}})$	*allocation*
D	::=		type table		|	$\%\mathsf{i} \leftarrow \%\mathsf{j}[\mathsf{k}]$	*field access*
		$(\ \mathsf{T}!\overline{\mathsf{ty}} <: \mathsf{A}\ )*$			|	$\%\mathsf{i} \leftarrow \%\mathsf{j}\ ?\ \mathsf{m}(\overline{\%\mathsf{k}}) : \%\mathsf{l}$	*dispatched call*
M	::=		method table		|	$\%\mathsf{i} \leftarrow \%\mathsf{j}\ ?\ \mathsf{m}!\overline{\mathsf{T}}(\overline{\%\mathsf{k}}) : \%\mathsf{l}$	*direct call*
		$\left(\ \langle \mathsf{m}!\overline{\mathsf{ty}'}\ \overline{\mathsf{st}},\ \overline{\mathsf{ty}}\rangle\ \right)*$		$i \in \mathbb{N},\ \mathsf{p} \in \mathbb{Z}$			

Fig. 5. Syntax of Jules

Types ty live in the immutable type table D, which contains both concrete (T) and abstract (A) types. Each type table entry is of the form $\mathsf{T}!\overline{\mathsf{ty}} <: \mathsf{A}$, introducing concrete type T, with fields of types $\overline{\mathsf{ty}}$, along with the single supertype A. Two predefined types are the concrete integer type Int, and the universal abstract supertype Any.

Method tables M contain method definitions of two sorts: first, original methods that come from source code; second, method instances compiled from original methods. To distinguish between

the two sorts, we store the type signature of the original method in every table entry. Thus, table entry $\langle \mathsf{m}!\overline{\mathsf{ty}'}\ \overline{\mathsf{st}},\ \overline{\mathsf{ty}}\rangle$ describes a method m with parameter types $\overline{\mathsf{ty}'}$ and the body comprised of instructions $\overline{\mathsf{st}}$; type signature $\overline{\mathsf{ty}}$ points to the original method. If $\overline{\mathsf{ty}}$ is equal to $\overline{\mathsf{ty}'}$, the entry defines an original method, and $\overline{\mathsf{st}}$ cannot contain direct calls.[3] Otherwise, $\overline{\mathsf{ty}'}$ denotes some concrete types $\overline{\mathsf{T}}$, and the entry defines a method instance compiled from $\mathsf{m}!\overline{\mathsf{ty}}$, specialized for concrete argument types $\overline{\mathsf{T}} <: \overline{\mathsf{ty}}$. Method table may contain multiple method definitions with the same name, but they have to have distinct type signatures.

Method bodies consist of instructions $\overline{\mathsf{st}}$. An instruction $\%\mathsf{i} \leftarrow \mathtt{op}$ consists of an operation $\mathtt{op}$ whose result is assigned to register $\%\mathsf{i}$. An instruction can assign from a primitive integer p, another register $\%\mathsf{j}$, a newly created struct $\mathsf{T}(\overline{\%\mathsf{k}})$ of type T with field values $\overline{\%\mathsf{k}}$, or the result of looking up a struct field as $\%\mathsf{j}[\mathsf{k}]$. Finally, the instruction may perform a function call. Calls can be dispatched, $\mathsf{m}(\overline{\%\mathsf{k}})$, where the target method is dynamically looked up, or they can be direct, $\mathsf{m}!\overline{\mathsf{T}}(\overline{\%\mathsf{k}})$, where the target method is specified. All calls are conditional: $\%\mathsf{j}$? `call` : $\%\mathsf{l}$, to allow for recursive functions. If the register $\%\mathsf{j}$ is non-zero, then `call` is performed. Otherwise, the result is the value of register $\%\mathsf{l}$. Conditional calls can be trivially transformed into unconditional calls; in examples, this transformation is performed implicitly.

4.2 Dynamic Semantics

Jules is parameterized over three components: method dispatch $\mathcal{D}$, type inference $\mathcal{I}$, and just-in-time compilation *jit*. We do not specify how the first two work, and merely provide their interface and a set of criteria that they must meet (in sections 4.2.2 and 4.2.3, respectively). The compiler, *jit*, is instantiated with either the identity function, which gives a regular non-optimizing semantics, or an optimizing compiler, which is defined in section 4.2.5. The optimizing compiler relies on type inference $\mathcal{I}$ to devirtualize method calls. Type inference also ensures well-formedness of method tables. Method dispatch $\mathcal{D}$ is used in operational semantics.

4.2.1 Operational Semantics. Fig. 6 gives rules for the dynamic semantics. Given a type table D as context, Jules can step a configuration F, M to F', M', written as $\mathsf{F}, \mathsf{M} \longrightarrow \mathsf{F}', \mathsf{M}'$. Stack frames F consist of a sequence of environment-instruction list pairs. Thus, $\mathsf{E}\ \overline{\mathsf{st}} \cdot \mathsf{F}$ denotes a stack with environment E and instructions $\overline{\mathsf{st}}$ on top, followed by a sequence of environment-instruction pairs. Each environment is a list of values $\mathsf{E} = \overline{\mathsf{v}}$, representing contents of the sequentially numbered registers. Environments can then be extended as $\mathsf{E} + \mathsf{v}$, indexed as $\mathsf{E}[\mathsf{k}]$, and their last value is $\mathit{last}(\mathsf{E})$ if E is not empty.

The small-step dynamic semantics is largely straightforward. The first four rules deal with register assignment: updating the environment with a constant value (Prim), the value in another register (Reg), a newly constructed struct (New), or the value in a field (Field). The remaining five rules deal with function calls, either dispatched $\mathsf{m}(\overline{\%\mathsf{k}})$ or direct $\mathsf{m}!\overline{\mathsf{T}}(\overline{\%\mathsf{k}})$. Call instructions are combined with conditioning: a call can only be made after testing the register $\%\mathsf{j}$, called a guard register. If the register value is zero, then the value of the alternate register $\%\mathsf{l}$ is returned (False1/False2). Otherwise, the call can proceed, by Disp for dispatched calls and Direct for direct ones. A dispatched call starts by prompting the JIT compiler to specialize method m from the method table M with the argument types $\overline{\mathsf{T}}$ and produce a new method table M'. Next, using the new table M', the dispatch mechanism $\mathcal{D}$ determines the method to invoke. Finally, the body $\overline{\mathsf{st}'}$ of the method and call arguments $\mathsf{E}[\overline{\mathsf{k}}]$ form a new stack frame for the callee, and the program steps with the extended stack and the new table. Direct calls are simpler because a direct call $\mathsf{m}!\overline{\mathsf{T}}$ uniquely identifies the method to invoke. Thus, the method's instructions are looked up in M by

[3] All function calls in Julia source code are dispatched calls.

$$v ::= p \mid T(\overline{v}) \qquad E ::= \overline{v} \qquad F ::= \epsilon \mid E\ \overline{st} \cdot F$$

$$\textsc{Prim}\ \frac{st = \%i \leftarrow p \qquad E' = E + p}{E\ st\ \overline{st} \cdot F,\ M \ \rightarrow\ E'\ \overline{st} \cdot F,\ M} \qquad \textsc{Reg}\ \frac{st = \%i \leftarrow \%j \qquad E' = E + E[j]}{E\ st\ \overline{st} \cdot F,\ M \ \rightarrow\ E'\ \overline{st} \cdot F,\ M} \qquad \textsc{New}\ \frac{st = \%i \leftarrow T(\overline{\%k}) \qquad E' = E + T(E[\overline{k}])}{E\ st\ \overline{st} \cdot F,\ M \ \rightarrow\ E'\ \overline{st} \cdot F,\ M}$$

$$\textsc{Field}\ \frac{st = \%i \leftarrow \%j[k] \qquad v = E[j] \qquad E' = E + v[k]}{E\ st\ \overline{st} \cdot F,\ M \ \rightarrow\ E'\ \overline{st} \cdot F,\ M} \qquad \textsc{False1}\ \frac{st = \%i \leftarrow \%j\ ?\ m(\overline{\%k}) : \%l \qquad 0 = E[j] \qquad E' = E + E[l]}{E\ st\ \overline{st} \cdot F,\ M \ \rightarrow\ E'\ \overline{st} \cdot F,\ M}$$

$$\textsc{False2}\ \frac{st = \%i \leftarrow \%j\ ?\ m!\overline{T}(\overline{\%k}) : \%l \qquad 0 = E[j] \qquad E' = E + E[l]}{E\ st\ \overline{st} \cdot F,\ M \ \rightarrow\ E'\ \overline{st} \cdot F,\ M}$$

$$\textsc{Disp}\ \frac{st = \%i \leftarrow \%j\ ?\ m(\overline{\%k}) : \%l \qquad 0 \neq E[j] \qquad \overline{T} = typeof(E[\overline{k}]) \qquad M' = jit(M, m, \overline{T}) \qquad \overline{st'} = body(\mathcal{D}(M', m, \overline{T})) \qquad E' = E[\overline{k}]}{E\ st\ \overline{st} \cdot F,\ M \ \rightarrow\ E'\ \overline{st'} \cdot E\ st\ \overline{st} \cdot F,\ M'}$$

$$\textsc{Direct}\ \frac{st = \%i \leftarrow \%j\ ?\ m!\overline{T}(\overline{\%k}) : \%l \qquad 0 \neq E[j] \qquad \overline{st'} = body(M[m!\overline{T}]) \qquad E' = E[\overline{k}]}{E\ st\ \overline{st} \cdot F,\ M \ \rightarrow\ E'\ \overline{st'} \cdot E\ st\ \overline{st} \cdot F,\ M}$$

$$\textsc{Ret}\ \frac{E'' = E + last(E')}{E'\ \epsilon \cdot E\ st\ \overline{st} \cdot F,\ M \ \rightarrow\ E''\ \overline{st} \cdot F,\ M}$$

Fig. 6. Dynamic semantics of Jules

the method signature, a new stack frame is created, and the program steps with the new stack and the same method table. The top frame without instructions to execute indicates the end of a function call (Ret): the last value of the top-frame environment becomes the return value of the call, and the top frame is popped from the stack.

Program execution begins with the frame ϵ main()[4], i.e. a call to the main function with an empty environment; the execution either diverges, finishes with a final configuration $E\ \epsilon$, or runs into an error. We define two notions of error. An *err* occurs only in the Disp rule, when the dispatch function $\mathcal{D}$ is undefined for the call; the *err* corresponds to a dynamic-dispatch error in Julia. A configuration is *wrong* if it cannot make a step for any other reason.

Definition 4.1 (Errors). A non-empty configuration F, M that cannot step $F, M \rightarrow F', M'$ has *erred* if its top-most frame, $E\ \overline{st}$, starts with $\%i \leftarrow \%j\ ?\ m(\overline{\%k}) : \%l$, where $\overline{T}$ is the types of $\overline{\%k}$ in E, $m \in M$, and $\mathcal{D}(M, m, \overline{T})$ is undefined. Otherwise, F, M is *wrong*.

4.2.2 Dispatch. Jules is parametric over method dispatch: any mechanism $\mathcal{D}$ that satisfies the *Dispatch Contract* (Def. 4.2) can be used. Julia's method dispatch mechanism is designed to, given method table, method name, and argument types, return the *most specific method applicable* to the given arguments if such a method exists and is unique. First, applicable methods are those whose declared type signature is a supertype of the argument type. Then, the most specific method is the one whose type signature is the most precise. Finally, only one most specific applicable method may exist, or else an error is produced. Each of these components appears in our dispatch definition. As in Julia, dispatch is only defined for tuples of concrete types.

[4]Recall that unconditional calls are implicitly expanded into conditional ones.

Definition 4.2 (Dispatch Contract). The *dispatch* function $\mathcal{D}(\mathsf{M}, \mathsf{m}, \overline{\mathsf{T}})$ takes method table M, method name m, and concrete argument types $\overline{\mathsf{T}}$, and returns a method $\mathsf{m}!\overline{\mathsf{ty}}\ \overline{\mathsf{st}} \in \mathsf{M}$ such that the following holds (we write $\overline{\mathsf{ty}} <: \overline{\mathsf{ty}'}$ as a shorthand for $\forall i.\ \mathsf{ty}_i <: \mathsf{ty}'_i$):

(1) $\overline{\mathsf{T}} <: \overline{\mathsf{ty}}$, meaning that $\mathsf{m}!\overline{\mathsf{ty}}$ is applicable to the given arguments;
(2) $\forall \mathsf{m}!\overline{\mathsf{ty}'}\ \overline{\mathsf{st}} \in \mathsf{M}.\ \overline{\mathsf{T}} <: \overline{\mathsf{ty}'} \implies \overline{\mathsf{ty}} <: \overline{\mathsf{ty}'}$, meaning that $\mathsf{m}!\overline{\mathsf{ty}}$ is the most specific applicable method.

4.2.3 Inference. The Julia compiler infers types of variables by forward data-flow analysis. Like dispatch, inference is complex, so we parameterize over it. For our purposes, an inference algorithm $\mathcal{I}$ returns a sound typing for a sequence of instructions in a given method table, $\mathcal{I}(\lfloor \mathsf{M} \rfloor, \overline{\mathsf{ty}}, \overline{\mathsf{st}}) = \overline{\mathsf{ty}'}$, where $\lfloor \mathsf{M} \rfloor$ denotes the table containing only methods without direct calls. Inference returns types $\overline{\mathsf{ty}'}$ such that each ty'_i is the type of register of st_i. Any inference algorithm that satisfies the Soundness and Monotonicity requirements is acceptable.

Requirement 4.1 (Soundness). *If $\mathcal{I}(\mathsf{M}, \overline{\mathsf{ty}}, \overline{\mathsf{st}}) = \overline{\mathsf{ty}'}$, then for any environment E compatible with $\overline{\mathsf{ty}}$, that is, typeof$(E_i) <: \mathsf{ty}_i$, and for any stack F, the following holds:*

(1) If $\mathsf{E}\ \overline{\mathsf{st}} \cdot \mathsf{F},\ \mathsf{M} \rightarrow^* \mathsf{F}' \cdot \mathsf{F}, \mathsf{M}'$ and cannot make another step, then $\mathsf{F}' \cdot \mathsf{F}, \mathsf{M}'$ has erred.
(2) If $\mathsf{E}\ \overline{\mathsf{st}} \cdot \mathsf{F},\ \mathsf{M} \rightarrow^* \mathsf{E}\ \mathsf{E}'\ \overline{\mathsf{st}''} \cdot \mathsf{F},\ \mathsf{M}'$, then $\overline{\mathsf{st}} = \overline{\mathsf{st}'}\ \overline{\mathsf{st}''}$ and typeof$(\mathsf{E}'_i) <: \mathsf{ty}'_i$.

The soundness requirement guarantees that if type inference succeeds on a method, then, when the method is called with compatible arguments, it will not enter a *wrong* state but may *err* at a dynamic call. Furthermore, if the method terminates, all its instructions evaluate to values compatible with the results of type inference. That is, when $\mathsf{E}\ \overline{\mathsf{st}} \cdot \mathsf{F},\ \mathsf{M} \rightarrow^* \mathsf{E}\ \mathsf{E}'\ \epsilon \cdot \mathsf{F},\ \mathsf{M}'$, we have typeof$(\mathsf{E}') <: \overline{\mathsf{ty}'}$.

The second requirement of type inference, monotonicity, is important to specialization: it guarantees that using more precise argument types for original method bodies succeeds and does not break assumptions of the caller about the callee's return type. If inference was not monotonic, then given more precise argument types, it could return a method specialization with a less precise return type. As a result, translating a dynamically dispatched call into a direct call may be unsound.

Requirement 4.2 (Monotonicity). *For all $\mathsf{M}, \overline{\mathsf{ty}}, \overline{\mathsf{st}}, \overline{\mathsf{ty}'}$, such that $\mathcal{I}(\mathsf{M}, \overline{\mathsf{ty}}, \overline{\mathsf{st}}) = \overline{\mathsf{ty}'}$,*

$$\forall\, \overline{\mathsf{ty}''}.\quad \overline{\mathsf{ty}''} <: \overline{\mathsf{ty}} \quad\implies\quad \exists \overline{\mathsf{ty}'''}.\ \ \mathcal{I}(\mathsf{M}, \overline{\mathsf{ty}''}, \overline{\mathsf{st}}) = \overline{\mathsf{ty}'''} \ \wedge\ \overline{\mathsf{ty}'''} <: \overline{\mathsf{ty}'}.$$

4.2.4 Well-Formedness. Initial Jules configuration ϵ main(), M is well-formed if the method table M is well-formed according to Def. 4.3. Such a configuration will either successfully terminate, *err*, or run forever, but it will never reach a *wrong* state.

Definition 4.3 (Well-Formedness of Method Table). A method table is *well-formed* $WF(\mathsf{M})$ if the following holds:

(1) Entry method $\mathsf{main}!\epsilon$ belongs to M.
(2) Every type ty in M, except Int and Any, is declared in D.
(3) Registers are numbered consecutively from 0, increasing by one for each parameter and instruction. An instruction assigning to %k only refers to registers %i such that $i < k$.
(4) For any original method $\langle \mathsf{m}!\overline{\mathsf{ty}}\ \overline{\mathsf{st}},\ \overline{\mathsf{ty}} \rangle \in \mathsf{M}$, the body is not empty and does not contain direct calls, and type inference succeeds $\mathcal{I}(\lfloor \mathsf{M} \rfloor, \overline{\mathsf{ty}}, \overline{\mathsf{st}}) = \overline{\mathsf{ty}'}$ on original methods $\lfloor \mathsf{M} \rfloor$.
(5) Any two methods in M with the same name, $\mathsf{m}!\overline{\mathsf{ty}}$ and $\mathsf{m}!\overline{\mathsf{ty}'}$, have distinct type signatures, i.e. $\overline{\mathsf{ty}} \neq \overline{\mathsf{ty}'}$.
(6) For any method specialization $\langle \mathsf{m}!\overline{\mathsf{ty}'}, \overline{\mathsf{ty}} \rangle \in \mathsf{M}$, i. e. $\overline{\mathsf{ty}'} \neq \overline{\mathsf{ty}}$, the following holds: $\overline{\mathsf{ty}'} = \overline{\mathsf{T}}$, and $\overline{\mathsf{T}} <: \overline{\mathsf{ty}}$, and $\langle \mathsf{m}!\overline{\mathsf{ty}}, \overline{\mathsf{ty}} \rangle \in \mathsf{M}$. Moreover, $\forall \mathsf{m}!\overline{\mathsf{ty}''} \in \mathsf{M}.\ \overline{\mathsf{T}} <: \overline{\mathsf{ty}''} \implies \overline{\mathsf{ty}} <: \overline{\mathsf{ty}''}$.

$$\frac{\begin{array}{c}\mathsf{m!}\overline{\mathsf{T}} \notin \mathsf{M} \qquad \overline{\mathsf{st}} = \mathit{body}(\mathcal{D}(\mathsf{M},\mathsf{m},\overline{\mathsf{T}})) \qquad \overline{\mathsf{ty'}} = \mathcal{I}(\lfloor \mathsf{M} \rfloor, \overline{\mathsf{T}}, \overline{\mathsf{st}}) \\ \overline{\mathsf{ty}} = \mathit{signature}(\mathcal{D}(\mathsf{M},\mathsf{m},\overline{\mathsf{T}})) \qquad \mathsf{M}_0 = \mathsf{M} + \langle \mathsf{m!}\overline{\mathsf{T}}\epsilon, \overline{\mathsf{ty}} \rangle \\ \overline{\mathsf{T}}\,\overline{\mathsf{ty'}} \vdash \mathsf{st}_0, \mathsf{M}_0 \rightsquigarrow \mathsf{st}'_0, \mathsf{M}_1 \quad \ldots \quad \overline{\mathsf{T}}\,\overline{\mathsf{ty'}} \vdash \mathsf{st}_n, \mathsf{M}_n \rightsquigarrow \mathsf{st}'_n, \mathsf{M}_{n+1} \qquad \mathsf{M}' = \mathsf{M}_{n+1}[\mathsf{m!}\overline{\mathsf{T}} \mapsto \langle \mathsf{m!}\overline{\mathsf{T}}\,\overline{\mathsf{st'}}, \overline{\mathsf{ty}} \rangle]\end{array}}{\mathit{jit}(\mathsf{M},\mathsf{m},\overline{\mathsf{T}}) = \mathsf{M}'}$$

$$\frac{\mathsf{m!}\overline{\mathsf{T}} \in \mathsf{M}}{\mathit{jit}(\mathsf{M},\mathsf{m},\overline{\mathsf{T}}) = \mathsf{M}} \qquad \frac{\begin{array}{c}\mathsf{st} = \%\mathsf{i} \leftarrow \%\mathsf{j}\ ?\ \mathsf{m}(\overline{\%\mathsf{k}}) : \%\mathsf{l} \qquad \overline{\mathsf{T}} = \overline{\mathsf{ty}}[\overline{\mathsf{k}}] \\ \mathsf{M}' = \mathit{jit}(\mathsf{M},\mathsf{m},\overline{\mathsf{T}}) \qquad \mathsf{st}' = \%\mathsf{i} \leftarrow \%\mathsf{j}\ ?\ \mathsf{m!}\overline{\mathsf{T}}(\overline{\%\mathsf{k}}) : \%\mathsf{l}\end{array}}{\overline{\mathsf{ty}} \vdash \mathsf{st}, \mathsf{M} \rightsquigarrow \mathsf{st}', \mathsf{M}'} \qquad \frac{\begin{array}{c}\mathsf{st} \neq \%\mathsf{i} \leftarrow \%\mathsf{j}\ ?\ \mathsf{m}(\overline{\%\mathsf{k}}) : \%\mathsf{l} \\ \text{or} \qquad \overline{\mathsf{T}} \neq \overline{\mathsf{ty}}[\overline{\mathsf{k}}]\end{array}}{\overline{\mathsf{ty}} \vdash \mathsf{st}, \mathsf{M} \rightsquigarrow \mathsf{st}, \mathsf{M}}$$

Fig. 7. Compilation: extending method table with a specialized method instance (top rule) and replacing a dynamically dispatched call with a direct method invocation in the extended table (bottom-right)

The last requirement ensures that only the most specific original methods have specializations, which precludes compilation from modifying program behavior. For example, consider type hierarchy `Int<:Number<:Any` and function `f` with original methods for `Any` and `Number`. If there are no compiled method instances, the call `f(5)` dispatches to `f(::Number)`. But if the method table contained a specialized instance `f(::Int)` of the original method `f(::Any)`, the call would dispatch to that instance, which is not related to the originally used `f(::Number)`. Thus, program behavior would be modified by compilation, which is undesired.

4.2.5 Compilation. Jules implements devirtualization through the $\mathit{jit}(\mathsf{M},\mathsf{m},\overline{\mathsf{T}})$ operation, shown in Fig. 7. The compiler specializes methods according to the inferred type information, replacing non-*err* dispatched calls with direct calls where possible. Compilation begins with some bookkeeping. First, it ensures that there is no pre-existing instance in the method table before compiling; otherwise, the table is returned immediately without modification, by the bottom-left rule. Next, using dispatch, it fetches the most applicable original method to compile. Then, using concrete argument types, the compiler runs the type inferencer on the method's body, producing an instruction typing $\overline{\mathsf{ty'}}$. Because the original method table is well-formed, monotonicity of $\mathcal{I}$ and the definition of $\mathcal{D}$ guarantee that type inference succeeds for $\overline{\mathsf{T}} <: \overline{\mathsf{ty}}$. Lastly, the compiler can begin translating instructions. Each instruction st_i is translated into an optimized instruction st'_i. This translation respects the existing type environment containing the argument types $\overline{\mathsf{T}}$ and instruction typing $\overline{\mathsf{ty'}}$. The translation $\overline{\mathsf{ty}} \vdash \mathsf{st}, \mathsf{M} \rightsquigarrow \mathsf{st}', \mathsf{M}'$ leaves all instructions unchanged except dispatched calls. Dispatched calls cause a recursive JIT invocation, followed by rewriting to a direct call. To avoid recursive compilation, a stub method $\langle \mathsf{m!}\overline{\mathsf{T}}\epsilon, \overline{\mathsf{ty}} \rangle$ is added at the beginning of compilation, and over-written when compilation is done. As the source program is finite and new types are not added during compilation, it terminates. Note that if the original method has concrete argument types, the compiler does nothing.

4.3 Type Groundedness and Stability

We now formally define the properties of interest, type stability and type groundedness. Recall the informal definition which stated that a function is type stable if its return type depends only on its argument types, and type grounded if every variable's type depends only on the argument types. In this definition, "types" really mean concrete types, as concrete types allow for optimizations. The following defines what it means for an original method to be type stable and type grounded.

Definition 4.4 (Stable and Grounded). Let $\mathsf{m!}\overline{\mathsf{ty}}\,\overline{\mathsf{st}}$ be an original method in a well-formed method table M. Given concrete argument types $\overline{\mathsf{T}} <: \overline{\mathsf{ty}}$, if $\mathcal{I}(\lfloor \mathsf{M} \rfloor, \mathsf{m}, \overline{\mathsf{T}}) = \overline{\mathsf{ty'}}$, and

$$\textsc{D-NoCall}\ \frac{\mathsf{st} \neq \%\mathsf{i} \leftarrow \%\mathsf{j}\ ?\ \mathsf{m}(\overline{\%\mathsf{k}}) : \%\mathsf{l} \qquad \mathsf{st} \neq \%\mathsf{i} \leftarrow \%\mathsf{j}\ ?\ \mathsf{m}!\overline{\mathsf{T}}(\overline{\%\mathsf{k}}) : \%\mathsf{l}}{\overline{\mathsf{ty}} \vdash^{\mathcal{D}}_{\mathsf{M}} \mathsf{st}}$$

$$\textsc{D-Disp}\ \frac{\mathsf{st} = \%\mathsf{i} \leftarrow \%\mathsf{j}\ ?\ \mathsf{m}(\overline{\%\mathsf{k}}) : \%\mathsf{l} \qquad \overline{\mathsf{T}} \neq \overline{\mathsf{ty}}[\overline{\mathsf{k}}]}{\overline{\mathsf{ty}} \vdash^{\mathcal{D}}_{\mathsf{M}} \mathsf{st}} \qquad \textsc{D-Direct}\ \frac{\mathsf{st} = \%\mathsf{i} \leftarrow \%\mathsf{j}\ ?\ \mathsf{m}!\overline{\mathsf{T}}(\overline{\%\mathsf{k}}) : \%\mathsf{l} \qquad \overline{\mathsf{T}} = \overline{\mathsf{ty}}[\overline{\mathsf{k}}] \qquad \mathsf{m}!\overline{\mathsf{T}} \in \mathsf{M}}{\overline{\mathsf{ty}} \vdash^{\mathcal{D}}_{\mathsf{M}} \mathsf{st}}$$

$$\textsc{D-Seq}\ \frac{\overline{\mathsf{ty}} \vdash^{\mathcal{D}}_{\mathsf{M}} \mathsf{st}_0 \qquad \ldots \qquad \overline{\mathsf{ty}} \vdash^{\mathcal{D}}_{\mathsf{M}} \mathsf{st}_n}{\overline{\mathsf{ty}} \vdash^{\mathcal{D}}_{\mathsf{M}} \overline{\mathsf{st}}}$$

$$\textsc{D-Table}\ \frac{\forall \langle \mathsf{m}!\overline{\mathsf{T}}\ \overline{\mathsf{st}'}, \overline{\mathsf{ty}} \rangle \in \mathsf{M}.\ \ \overline{\mathsf{T}} \neq \overline{\mathsf{ty}} \ \wedge\ \overline{\mathsf{st}'} \neq \epsilon \quad \Longrightarrow \quad \mathsf{m}!\overline{\mathsf{ty}}\ \overline{\mathsf{st}} \in \mathsf{M} \ \wedge\ \overline{\mathsf{ty}'} = \mathcal{I}(\lfloor \mathsf{M} \rfloor, \overline{\mathsf{T}}, \overline{\mathsf{st}}) \ \wedge\ \overline{\mathsf{T}}\,\overline{\mathsf{ty}'} \vdash^{\mathcal{D}}_{\mathsf{M}} \overline{\mathsf{st}'}}{\vdash^{\mathcal{D}} \mathsf{M}}$$

Fig. 8. Maximal devirtualization of instructions and method tables

(1) $last(\overline{\mathsf{ty}'}) = \mathsf{T}'$, i.e. the return type is concrete, then the method is *type stable for* $\overline{\mathsf{T}}$,
(2) $\overline{\mathsf{ty}'} = \overline{\mathsf{T}'}$, i.e. all register types are concrete, then the method is *type-grounded for* $\overline{\mathsf{T}}$.

Furthermore, a method is called *type stable (grounded) for a set W of concrete argument types* if it is type stable (grounded) for every $\overline{\mathsf{T}} \in W$.

As all method instances are compiled from some original method definitions, type stabiltiy and groundedness for instances is defined in terms of their originals.

Definition 4.5. A method instance $\langle \mathsf{m}!\overline{\mathsf{T}}\ \overline{\mathsf{st}'},\ \overline{\mathsf{ty}} \rangle$ is called *type stable (grounded)*, if its original method $\mathsf{m}!\overline{\mathsf{ty}}$ is type stable (grounded) for $\overline{\mathsf{T}}$.

4.3.1 Full Devirtualization. The key property of type groundedness is that compiling a type-grounded method results in a fully devirtualized instance. We say that a method instance $\mathsf{m}!\overline{\mathsf{T}}\ \overline{\mathsf{st}}$ is *fully devirtualized* if $\overline{\mathsf{st}}$ does not contain any dispatched calls. To show that *jit* indeed has the above property, we will need an additional notion, *maximal devirtualization*, which is defined in Fig. 8. Intuitively, the predicate $\overline{\mathsf{ty}} \vdash^{\mathcal{D}}_{\mathsf{M}} \mathsf{st}$ states that an instruction st does not contain a dispatched call that can be resolved in table M for argument types found in $\overline{\mathsf{ty}}$. Then, a method instance is maximally devirtualized if this predicate holds for every instruction using $\overline{\mathsf{ty}}$ that combines argument types with the results of type inference.

Next, we review the definition from Fig. 8 in more details. D-NoCall states that an instruction without a call is maximally devirtualized. D-Disp requires that for a dispatched call, $\overline{\mathsf{ty}}$ does not have precise enough type information to resolve the call with $\mathcal{D}$. Finally, D-Direct allows a direct call to a concrete method with the right type signature: as concrete types do not have subtypes, the $\mathsf{m}!\overline{\mathsf{T}}$ with concrete argument types is exactly the definition that a call $\mathsf{m}(\overline{\%\mathsf{k}})$ would dispatch to. D-Seq simply checks that all instructions in a sequence $\overline{\mathsf{st}}$ are devirtualized in the same register typing $\overline{\mathsf{ty}}$. Here, $\overline{\mathsf{ty}}$ has to contain typing for all instructions st_i, because later instructions refer to the registers of the previous ones. The last rule $\vdash^{\mathcal{D}} \mathsf{M}$ covers the entire table M, requiring all methods to be maximally devirtualized. Namely, D-Table says that for all method instances (condition $\overline{\mathsf{T}} \neq \overline{\mathsf{ty}}$ implies that $\mathsf{m}!\overline{\mathsf{T}}$ is not an original method) that are not stubs ($\overline{\mathsf{st}'} \neq \epsilon$), the body

$$\textsc{C-NoDisp}\ \frac{\mathsf{st} \neq \%\mathsf{i} \leftarrow \%\mathsf{j}\ ?\ \mathsf{m}(\overline{\%\mathsf{k}}) : \%\mathsf{l}}{\overline{\mathsf{ty}} \vdash \mathsf{st}, M \rightsquigarrow \mathsf{st}, M} \qquad \textsc{C-Disp}\ \frac{\mathsf{st} = \%\mathsf{i} \leftarrow \%\mathsf{j}\ ?\ \mathsf{m}(\overline{\%\mathsf{k}}) : \%\mathsf{l} \qquad \overline{\mathsf{T}} \neq \overline{\mathsf{ty}}[\overline{\mathsf{k}}]}{\overline{\mathsf{ty}} \vdash \mathsf{st}, M \rightsquigarrow \mathsf{st}, M}$$

$$\textsc{C-Direct}\ \frac{\mathsf{st} = \%\mathsf{i} \leftarrow \%\mathsf{j}\ ?\ \mathsf{m}(\overline{\%\mathsf{k}}) : \%\mathsf{l} \qquad \overline{\mathsf{T}} = \overline{\mathsf{ty}}[\overline{\mathsf{k}}] \qquad \mathsf{m}!\overline{\mathsf{T}} \in M \qquad \mathsf{st}' = \%\mathsf{i} \leftarrow \%\mathsf{j}\ ?\ \mathsf{m}!\overline{\mathsf{T}}(\overline{\%\mathsf{k}}) : \%\mathsf{l}}{\overline{\mathsf{ty}} \vdash \mathsf{st}, M \rightsquigarrow \mathsf{st}', M}$$

$$\textsc{C-Instance}\ \frac{\begin{array}{c}\mathsf{st} = \%\mathsf{i} \leftarrow \%\mathsf{j}\ ?\ \mathsf{m}(\overline{\%\mathsf{k}}) : \%\mathsf{l} \qquad \overline{\mathsf{T}} = \overline{\mathsf{ty}}[\overline{\mathsf{k}}] \qquad \mathsf{m}!\overline{\mathsf{ty}'}\,\overline{\mathsf{st}} = \mathcal{D}(M, \mathsf{m}, \overline{\mathsf{T}}) \qquad \overline{\mathsf{T}} \neq \overline{\mathsf{ty}'} \\ \overline{\mathsf{ty}''} = \mathcal{I}(\lfloor M \rfloor, \overline{\mathsf{T}}, \overline{\mathsf{st}}) \qquad M_0 = M + \langle \mathsf{m}!\overline{\mathsf{T}}\epsilon, \overline{\mathsf{ty}'} \rangle \\ \overline{\mathsf{T}}\,\overline{\mathsf{ty}''} \vdash \mathsf{st}_0, M_0 \rightsquigarrow \mathsf{st}'_0, M_1 \quad \ldots \quad \overline{\mathsf{T}}\,\overline{\mathsf{ty}''} \vdash \mathsf{st}_n, M_n \rightsquigarrow \mathsf{st}'_n, M_{n+1} \\ \mathsf{st}' = \%\mathsf{i} \leftarrow \%\mathsf{j}\ ?\ \mathsf{m}!\overline{\mathsf{T}}(\overline{\%\mathsf{k}}) : \%\mathsf{l} \qquad M' = M_{n+1} + \langle \mathsf{m}!\overline{\mathsf{T}}\,\overline{\mathsf{st}'}, \overline{\mathsf{ty}'} \rangle\end{array}}{\overline{\mathsf{ty}} \vdash \mathsf{st}, M \rightsquigarrow \mathsf{st}', M'}$$

Fig. 9. Reformulated compilation

$\overline{\mathsf{st}'}$ is maximally devirtualized according to the typing of the original method with respect to the instance's argument types.

Using the notion of maximal devirtualization, we can connect type groundedness and full devirtualization.

Lemma 4.6 (Full Devirtualization). *If* M *is well-formed and maximally devirtualized, then any type-grounded method instance* $\mathsf{m}!\overline{\mathsf{T}}\,\overline{\mathsf{st}'} \in M$ *is fully devirtualized.*

Proof. Follows from the definitions of type groundedness and maximal devirtualization; details can be found in the extended version [Pelenitsyn et al. 2021b]. □

The final step is to show that compilation defined in Fig. 7 preserves maximal devirtualization. To simplify the proof, Fig. 9 reformulates Fig. 7 by inlining *jit* into the compilation relation. The relation does not have a rule for processing direct calls because we compile only original methods. Since the set of method instances is finite, the relation is well-defined: every recursive call to compilation happens for a method table that contains at least one more instance. Every compilation step produces a maximally devirtualized instruction and potentially extends the method table with new maximally devirtualized instances.

Lemma 4.7 (Preserving Well Formedness). *For any method table* M *that is well-formed* $WF(M)$*, register typing* $\overline{\mathsf{ty}}$*, and instruction* st*, if the instruction* st *is compiled,* $\overline{\mathsf{ty}} \vdash \mathsf{st}, M \rightsquigarrow \mathsf{st}', M'$*, then the new table is well-formed* $WF(M')$*.*

Proof. By induction on the derivation of $\overline{\mathsf{ty}} \vdash \mathsf{st}, M \rightsquigarrow \mathsf{st}', M'$. The only case that modifies the method table is C-Instance. There, by analyzing M_0, we can see that it is well-formed, and thus the induction hypothesis can be applied to $\overline{\mathsf{T}}\,\overline{\mathsf{ty}''} \vdash \mathsf{st}_0, M_0 \rightsquigarrow \mathsf{st}'_0, M_1$, and then to all $\overline{\mathsf{T}}\,\overline{\mathsf{ty}''} \vdash \mathsf{st}_i, M_i \rightsquigarrow \mathsf{st}'_i, M_{i+1}$. Since M_{n+1} is well-formed, so is M'. More details are available in the extended version [Pelenitsyn et al. 2021b]. □

Lemma 4.8 (Preserving Maximal Devirtualization). *For any well-formed method table* M*, register typing* $\overline{\mathsf{ty}}$*, and instruction* st*, if the method table is maximally devirtualized,* $\vdash^{\mathcal{D}} M$*, and the instruction* st *is compiled,* $\overline{\mathsf{ty}} \vdash \mathsf{st}, M \rightsquigarrow \mathsf{st}', M'$*, then the following holds:*

(1) the resulting instruction is maximally devirtualized in the new table, $\overline{\mathsf{ty}} \vdash^{\mathcal{D}}_{M'} \overline{\mathsf{st}'}$*,*

(2) the new table is maximally devirtualized, $\vdash^{\mathcal{D}} \mathsf{M}'$,
(3) any maximally devirtualized instruction stays maximally devirtualized in the new table, $\forall \overline{\mathsf{ty}^x}, \overline{\mathsf{st}^x}.\quad \overline{\mathsf{ty}^x} \vdash^{\mathcal{D}}_{\mathsf{M}} \overline{\mathsf{st}^x} \implies \overline{\mathsf{ty}^x} \vdash^{\mathcal{D}}_{\mathsf{M}'} \overline{\mathsf{st}^x}$.

Proof. By induction on the derivation of $\overline{\mathsf{ty}} \vdash \mathsf{st}, \mathsf{M} \rightsquigarrow \mathsf{st}', \mathsf{M}'$. The most interesting case is C-Instance where compilation of additional instances happens. The key step of the proof is to observe that the induction hypothesis is applicable to $\overline{\mathsf{T}}\,\overline{\mathsf{ty}''} \vdash \mathsf{st}_0, \mathsf{M}_0 \rightsquigarrow \mathsf{st}'_0, \mathsf{M}_1$, and then to all consecutive $\overline{\mathsf{T}}\,\overline{\mathsf{ty}''} \vdash \mathsf{st}_i, \mathsf{M}_i \rightsquigarrow \mathsf{st}'_i, \mathsf{M}_{i+1}$, which is possible due to Lem. 4.7 and facts (2). Using facts (3), we propagate the information about maximal devirtualization of st_i in their respective tables to the final M'. A complete proof is available in [Pelenitsyn et al. 2021b]. □

Putting it all together, we have shown that type-grounded methods compile to method instances without dynamically dispatched calls. Namely, since a well-formed original table is trivially maximally devirtualized, and compilation preserves this property by Lem. 4.8, executing a well-formed program results in a method table where all method instances are maximally devirtualized. Furthermore, type-grounded instances are fully devirtualized by Lem. 4.6, and thus do not contain dispatched calls.

Theorem 4.9 (Grounded Methods Never Dispatch). *For any initial well-formed* M, *if*

$$\epsilon\ \mathsf{main}(),\ \mathsf{M} \rightarrow^* \mathsf{E}\ (\%\mathsf{i} \leftarrow \%\mathsf{j}\ ?\ \mathsf{m}!\overline{\mathsf{T}}(\overline{\%\mathsf{k}}) : \%\mathsf{l})\ \overline{\mathsf{st}} \cdot \mathsf{F},\ \mathsf{M}' \quad \wedge \quad \mathsf{m}!\overline{\mathsf{T}} \textit{ is type-grounded in } \mathsf{M}',$$

then $\mathsf{m}!\overline{\mathsf{T}}$ *is fully devirtualized in* M'.

4.3.2 Relationship between Stability and Groundedness. While it is type groundedness that enables devirtualization, the weaker property, type stability, is also important in practice. Namely, stability of calles is crucial for groundedness of the caller if the type inference algorithm analyzes function calls using nothing but type information about the arguments. (A formal definition and a proof of this property can be found in the extended version [Pelenitsyn et al. 2021b].) A more powerful type inference algorithm might be able to work around unstable methods in some cases, but even then, stability would be needed in general.

As an example, consider two type-unstable callees that return either an integer or a string depending on the argument value, `f(x) = (x>0) ? 1 : "1"` and `g(x) = (rand()>x)? 1 : "1"`, and two calls, `f(y)` and `g(y)`, where `y` is equal to 0.5. If only type information about `y` is available to type inference, both `f(y)` and `g(y)` are deemed to have abstract return types. Therefore, the results of such function calls cannot be stored in concretely typed registers, which immediately makes the calling code ungrounded. If type inference were to analyze the value of `y` (not just its type), the result of `f(y)` could be stored in a concrete, integer-valued register, as only the first branch of `f` is known to execute. However, the other call, `g(y)`, would still have to be assigned to an abstract register. Thus, to enable groundedness and optimization of the client code regardless of the value of argument `x`, `f(x)` and `g(x)` need to be type-stable.

Note that stability is a necessary but not sufficient condition for groundedness of the client, as conditional branches may lead to imprecise types, like in the `f0` example from Fig. 3.

4.4 Correctness of Compilation

In this section, we prove that evaluating a program with just-in-time compilation yields the same result as if no compilation occurred. We use two instantiations of the Jules semantics, one (written $\rightarrow_{\mathcal{D}}$) where *jit* is the identity, and the other (written $\rightarrow_{\mathsf{JIT}}$) where *jit* is defined as before. Our proof strategy is to

(1) define the optimization relation $\triangleright$ which relates original and optimized code (Fig. 10),

$$\textsc{OI-Refl}\ \frac{}{\overline{\mathsf{ty}} \vdash_{\mathsf{M},\mathsf{M}'} \mathsf{st} \triangleright \mathsf{st}}$$

$$\textsc{OI-Direct}\ \frac{\mathsf{st} = \%\mathsf{i} \leftarrow \%\mathsf{j}\ ?\ \mathsf{m}(\overline{\%\mathsf{k}}) : \%\mathsf{l} \qquad \overline{\mathsf{T}} = \overline{\mathsf{ty}}[\overline{\mathsf{k}}] \quad \mathit{signtr}(\mathcal{D}(\mathsf{M}, \mathsf{m}, \overline{\mathsf{T}})) = \mathit{o\text{-}signtr}(\mathsf{M}'[\mathsf{m}!\overline{\mathsf{T}}]) \quad \mathsf{st}' = \%\mathsf{i} \leftarrow \%\mathsf{j}\ ?\ \mathsf{m}!\overline{\mathsf{T}}(\overline{\%\mathsf{k}}) : \%\mathsf{l}}{\overline{\mathsf{ty}} \vdash_{\mathsf{M},\mathsf{M}'} \mathsf{st} \triangleright \mathsf{st}'}$$

$$\textsc{OI-Seq}\ \frac{\overline{\mathsf{ty}} \vdash_{\mathsf{M},\mathsf{M}'} \mathsf{st}_0 \triangleright \mathsf{st}'_0 \quad \ldots \quad \overline{\mathsf{ty}} \vdash_{\mathsf{M},\mathsf{M}'} \mathsf{st}_n \triangleright \mathsf{st}'_n}{\overline{\mathsf{ty}} \vdash_{\mathsf{M},\mathsf{M}'} \overline{\mathsf{st}} \triangleright \overline{\mathsf{st}'}}$$

$$\textsc{O-Table}\ \frac{\mathsf{M} = \lfloor \mathsf{M}' \rfloor \qquad \forall \langle \mathsf{m}!\overline{\mathsf{T}}\,\overline{\mathsf{st}'},\ \overline{\mathsf{ty}} \rangle, \langle \mathsf{m}!\overline{\mathsf{ty}}\,\overline{\mathsf{st}},\ \overline{\mathsf{ty}} \rangle \in \mathsf{M}',\ \text{s.t. } \overline{\mathsf{T}} \neq \overline{\mathsf{ty}},\ \overline{\mathsf{st}'} \neq \epsilon.\ \overline{\mathsf{ty}'} = \mathcal{I}(\mathsf{M}, \overline{\mathsf{T}}, \overline{\mathsf{st}}) \qquad \overline{\mathsf{T}}\,\overline{\mathsf{ty}'} \vdash_{\mathsf{M},\mathsf{M}'} \overline{\mathsf{st}} \triangleright \overline{\mathsf{st}'}}{\mathsf{M} \triangleright \mathsf{M}'}$$

$$\Delta \quad ::= \quad \epsilon \mid \overline{\mathsf{ty}} \cdot \Delta' \qquad \text{stack typing}$$

$$\textsc{O-StackEmpty}\ \frac{}{\epsilon \vdash_{\mathsf{M},\mathsf{M}'} \epsilon \triangleright \epsilon} \qquad \textsc{O-Stack}\ \frac{\overline{\mathsf{ty}} \vdash_{\mathsf{M},\mathsf{M}'} \overline{\mathsf{st}} \triangleright \overline{\mathsf{st}'} \qquad \Delta \vdash_{\mathsf{M},\mathsf{M}'} \mathsf{F} \triangleright \mathsf{F}'}{\overline{\mathsf{ty}} \cdot \Delta \vdash_{\mathsf{M},\mathsf{M}'} (\mathsf{E}\,\overline{\mathsf{st}}) \cdot \mathsf{F} \triangleright (\mathsf{E}\,\overline{\mathsf{st}'}) \cdot \mathsf{F}'}$$

$$\textsc{I-StackEmpty}\ \frac{}{\vdash^{\mathcal{I}}_{\mathsf{M}} \epsilon <: \epsilon}$$

$$\textsc{I-Stack}\ \frac{\exists \mathsf{m}!\overline{\mathsf{ty}'}\,\overline{\mathsf{st}_b} \in \mathsf{M}. \qquad \mathsf{E} = \mathsf{E}'\mathsf{E}'' \qquad \mathsf{F} \neq \epsilon \implies \mathsf{F}, \mathsf{M} \rightarrow_{\mathcal{D}} \mathsf{E}'\,\overline{\mathsf{st}_b} \cdot \mathsf{F}, \mathsf{M} \quad \mathsf{E}'\,\overline{\mathsf{st}_b} \cdot \mathsf{F}, \mathsf{M} \rightarrow^*_{\mathcal{D}} \mathsf{E}\,\overline{\mathsf{st}} \cdot \mathsf{F}, \mathsf{M} \qquad \overline{\mathsf{T}} = \mathit{typeof}(\mathsf{E}') \qquad \overline{\mathsf{ty}''} = \mathcal{I}(\mathsf{M}, \overline{\mathsf{T}}, \overline{\mathsf{st}_b}) \quad \overline{\mathsf{T}} <: \overline{\mathsf{ty}'} \qquad \overline{\mathsf{T}}\,\overline{\mathsf{ty}''} <: \overline{\mathsf{ty}} \qquad \vdash^{\mathcal{I}}_{\mathsf{M}} \mathsf{F} <: \Delta}{\vdash^{\mathcal{I}}_{\mathsf{M}} (\mathsf{E}\,\overline{\mathsf{st}}) \cdot \mathsf{F} <: \overline{\mathsf{ty}} \cdot \Delta}$$

$$\textsc{O-Config}\ \frac{\vdash^{\mathcal{I}}_{\mathsf{M}} \mathsf{F} <: \Delta \qquad \Delta \vdash_{\mathsf{M},\mathsf{M}'} \mathsf{F} \triangleright \mathsf{F}' \qquad \mathsf{M} \triangleright \mathsf{M}'}{\mathsf{F}, \mathsf{M} \triangleright \mathsf{F}', \mathsf{M}' <: \Delta}$$

Fig. 10. Optimization relation for instructions, method tables, stacks, and configurations

(2) show that compilation from Fig. 9 does produce optimized code (Thm. 4.12),
(3) show that evaluating a program with $\rightarrow_{\mathsf{JIT}}$ or $\rightarrow_{\mathcal{D}}$ gives the same result (Thm. 4.15).

The main Thm. 4.15 is a corollary of bisimulation between $\rightarrow_{\mathcal{D}}$ and $\rightarrow_{\mathsf{JIT}}$ (Lem. 4.14). The bisimulation lemma uses Thm. 4.12 but otherwise, it has a proof similar to the proof of bisimulation between original and optimized code both running with $\rightarrow_{\mathcal{D}}$ (Lem. 4.10). A corollary of the latter bisimulation lemma, Thm. 4.11, shows the correctness of *jit* as an ahead-of-time compilation strategy.

First, Fig. 10 defines optimization relations for instructions, method tables, stacks, and configurations. According to the definition of $\mathsf{M} \triangleright \mathsf{M}'$, method table M' optimizes M if (1) it has all the same original methods[5], and (2) bodies of compiled method instances optimize the original methods using more specific type information available to the instance. These requirements guarantee that dispatching in an original and optimized method tables will always resolve to related methods. Optimization of instructions only allows for replacing dynamically dispatched calls with direct calls in the optimized table. Optimization of stacks ensures that for all frames, instructions are optimized accordingly, and requires all value environments to coincide. The first premise of configuration optimization O-Config guarantees that the original configuration F, M is obtained by calling

[5] $\lfloor \mathsf{M}' \rfloor$ denotes the method table containing all original methods of M' without compiled instances.

methods from the original method table M, and bodies of those methods are amenable to type inference. Based on the results of type inference, stacks F, F' need to be related in method tables M, M', and the method tables themselves also need to be related.

As we show below, when run with the dispatch semantics, related configurations are guaranteed to run in lock-step and produce the same result.

Lemma 4.10 (Bisimulation of Related Configurations). *For any well-formed method tables* M *and* M' *(i.e.* $WF(\mathsf{M})$, $WF(\mathsf{M}')$ *according to Def. 4.3) where* M' *does not have stubs (i.e. all method bodies* $\neq \epsilon$*), any stacks* F_1, F'_1, *and stack typing* Δ_1 *that relates the configurations,* $\mathsf{F}_1, \mathsf{M} \triangleright \mathsf{F}'_1, \mathsf{M}' <: \Delta_1$, *the following holds:*

(1) Forward direction:

$$\mathsf{F}_1, \mathsf{M} \to_{\mathcal{D}} \mathsf{F}_2, \mathsf{M} \implies \exists \mathsf{F}'_2, \Delta'_1. \quad \mathsf{F}'_1, \mathsf{M}' \to_{\mathcal{D}} \mathsf{F}'_2, \mathsf{M}' \ \wedge \ \mathsf{F}_2, \mathsf{M} \triangleright \mathsf{F}'_2, \mathsf{M}' <: \Delta'_1.$$

(2) Backward direction:

$$\mathsf{F}'_1, \mathsf{M}' \to_{\mathcal{D}} \mathsf{F}'_2, \mathsf{M}' \implies \exists \mathsf{F}_2, \Delta'_1. \quad \mathsf{F}_1, \mathsf{M} \to_{\mathcal{D}} \mathsf{F}_2, \mathsf{M} \ \wedge \ \mathsf{F}_2, \mathsf{M} \triangleright \mathsf{F}'_2, \mathsf{M}' <: \Delta'_1.$$

Proof. For both directions, the proof goes by case analysis on the optimization relation for configurations and case analysis on the step of execution. The most interesting cases are the ones where F_1, M steps by Disp, with $\mathsf{F}'_1, \mathsf{M}'$ stepping by Disp or Direct. The key observation is that because of well-formedness of method tables and $\mathsf{M} \triangleright \mathsf{M}'$, the optimized configuration $\mathsf{F}'_1, \mathsf{M}'$ correctly dispatches or directly invokes a specialized method instance of the same original method that F_1, M dispatches to. A complete proof is available in the extended version of the paper [Pelenitsyn et al. 2021b]. □

Theorem 4.11 (Correctness of Optimized Method Table). *For any well-formed method tables* M *and* M' *where (1)* $\mathsf{M} \triangleright \mathsf{M}'$, *(2) table* M' *does not have stubs, and (3)* $\langle \mathsf{main}!\epsilon\ \overline{\mathsf{st}}, \epsilon \rangle \in \mathsf{M}$,

$$\epsilon\ \overline{\mathsf{st}}, \mathsf{M} \to^*_{\mathcal{D}} \mathsf{E}\ \epsilon, \mathsf{M} \iff \epsilon\ \overline{\mathsf{st}}, \mathsf{M}' \to^*_{\mathcal{D}} \mathsf{E}\ \epsilon, \mathsf{M}',$$

i.e. program $\overline{\mathsf{st}}$ *runs to the same final value environment in both tables.*

Proof. This is a corollary of Lem. 4.10; detailes are in [Pelenitsyn et al. 2021b]. □

Next, we show that compilation as defined in Fig. 9 produces optimized code in an optimized method table according to Fig. 10.

Theorem 4.12 (Compilation Satisfies Optimization Relation). *For any well-formed method tables* M *and* M', *typing* $\overline{\mathsf{ty}}$, *and instruction* st, *such that*

$$\mathsf{M} \triangleright \mathsf{M}' \ \wedge \ \overline{\mathsf{ty}} \vdash \mathsf{st}, \mathsf{M}' \rightsquigarrow \mathsf{st}'', \mathsf{M}'',$$

it holds that:

(1) $\overline{\mathsf{ty}} \vdash_{\mathsf{M},\mathsf{M}''} \mathsf{st} \triangleright \mathsf{st}''$,
(2) $\mathsf{M} \triangleright \mathsf{M}''$,
(3) and the optimization relation on M, M' *is preserved on* M, M''*:*
$\forall \overline{\mathsf{ty}^x}, \mathsf{st}^x, \mathsf{st}^y. \quad \overline{\mathsf{ty}^x} \vdash_{\mathsf{M},\mathsf{M}'} \mathsf{st}^x \triangleright \mathsf{st}^y \implies \overline{\mathsf{ty}^x} \vdash_{\mathsf{M},\mathsf{M}''} \mathsf{st}^x \triangleright \mathsf{st}^y.$

Proof. By induction on the derivation of $\overline{\mathsf{ty}} \vdash \mathsf{st}, \mathsf{M}' \rightsquigarrow \mathsf{st}'', \mathsf{M}''$. Similar to the proof of Lem. 4.8 on maximal devirtualization, the most interesting case is C-Instance where compilation of additional instances happens. The key step of the proof is to observe that the induction hypothesis is applicable to $\overline{\mathsf{T}}\,\overline{\mathsf{ty}''} \vdash \mathsf{st}_0, \mathsf{M}_0 \rightsquigarrow \mathsf{st}'_0, \mathsf{M}_1$, and then, using facts (2) and Lem. 4.7, to all $\overline{\mathsf{T}}\,\overline{\mathsf{ty}''} \vdash \mathsf{st}_i, \mathsf{M}_i \rightsquigarrow \mathsf{st}'_i, \mathsf{M}_{i+1}$. Using facts (3), we can propagate the information about optimization of $\mathsf{st}_i, \mathsf{st}'_i$ and respective tables M_i to the final M''. A complete proof is available in the extended version [Pelenitsyn et al. 2021b]. □

Finally, using an auxiliary lemma about preserving stub methods during compilation, we can show that the JIT-compilation semantics is equivalent to the dispatch semantics.

Lemma 4.13 (Preserving Stubs). *For any well-formed method tables* M' *and* M''*, typing* $\overline{\mathsf{ty}}$*, and instruction* st*, such that*

$$\overline{\mathsf{ty}} \vdash \mathsf{st}, \mathsf{M}' \rightsquigarrow \mathsf{st}'', \mathsf{M}'',$$

it holds that

$$\{\overline{\mathsf{T}} \mid \mathsf{m}!\overline{\mathsf{T}}\,\epsilon \in \mathsf{M}'\} = \{\overline{\mathsf{T}} \mid \mathsf{m}!\overline{\mathsf{T}}\,\epsilon \in \mathsf{M}''\},$$

i.e. the set of stubbed method instances is preserved by a compilation step.

Proof. By induction on the derivation of $\overline{\mathsf{ty}} \vdash \mathsf{st}, \mathsf{M}' \rightsquigarrow \mathsf{st}'', \mathsf{M}''$, similar to the proof of Lem. 4.7 on well formedness. See [Pelenitsyn et al. 2021b]. □

Lemma 4.14 (Bisimulation of Related Configurations with Dispatch and JIT Semantics). *For any well-formed method tables* M *and* M' *where* M' *does not have stubs, any frame stacks* F_1, F'_1, *and stack typing* Δ_1*, such that* $\mathsf{F}_1, \mathsf{M} \triangleright \mathsf{F}'_1, \mathsf{M}' <: \Delta_1$*, the following holds:*

(1) Forward direction:

$$\mathsf{F}_1, \mathsf{M} \to_{\mathcal{D}} \mathsf{F}_2, \mathsf{M} \implies \exists \mathsf{F}'_2, \mathsf{M}'', \Delta'_1.\ \ \mathsf{F}'_1, \mathsf{M}' \to_{\mathsf{JIT}} \mathsf{F}'_2, \mathsf{M}'' \ \wedge\ \mathsf{F}_2, \mathsf{M} \triangleright \mathsf{F}'_2, \mathsf{M}'' <: \Delta'_1.$$

(2) Backward direction:

$$\mathsf{F}'_1, \mathsf{M}' \to_{\mathsf{JIT}} \mathsf{F}'_2, \mathsf{M}'' \implies \exists \mathsf{F}_2, \Delta'_1.\ \ \mathsf{F}_1, \mathsf{M} \to_{\mathcal{D}} \mathsf{F}_2, \mathsf{M} \ \wedge\ \mathsf{F}_2, \mathsf{M} \triangleright \mathsf{F}'_2, \mathsf{M}'' <: \Delta'_1.$$

Furthermore, M'' *is well-formed and does not have stubs.*

Proof. By case analysis on the derivation of optimization $\mathsf{F}_1, \mathsf{M} \triangleright \mathsf{F}'_1, \mathsf{M}' <: \Delta_1$ and case analysis on the step ($\to_{\mathcal{D}}$ for the forward and $\to_{\mathsf{JIT}}$ for the backward direction), similarly to the proof of bisimulation for the dispatch semantics (Lem. 4.10). The only difference appears in cases where $\mathsf{F}'_1, \mathsf{M}'$ steps by Disp: these are the only places where JIT compilation fires and M'' might be different from M'. The key observation is that compilation produces M'' which (1) optimizes M by Thm. 4.12, $\mathsf{M} \triangleright \mathsf{M}''$, (2) is well-formed by Lem. 4.7, and (3) does not contain stubs by Lem. 4.13. More details are available in the extended version of the paper [Pelenitsyn et al. 2021b]. □

Theorem 4.15 (Correctness ot JIT). *For any original well-formed method table* M *the following holds:*

$$\epsilon\ \overline{\mathsf{st}}, \mathsf{M} \to^*_{\mathcal{D}} \mathsf{E}\ \epsilon, \mathsf{M} \iff \epsilon\ \overline{\mathsf{st}}, \mathsf{M} \to^*_{\mathsf{JIT}} \mathsf{E}\ \epsilon, \mathsf{M}',$$

i.e. program $\overline{\mathsf{st}}$ *runs to the final environment* E *with the dispatch semantics* $\to_{\mathcal{D}}$ *if and only if it runs to the same environment with the JIT-compilation semantics* $\to_{\mathsf{JIT}}$*.*

Proof. This is a corollary of Lem. 4.14. Be reflexivity of the optimization relation and well-formedness of M, we know: $\overline{\mathsf{st}}, \mathsf{M} \triangleright \overline{\mathsf{st}}, \mathsf{M} <: \overline{\mathsf{ty}}$, where $\overline{\mathsf{ty}} = \mathcal{I}(\mathsf{M}, \epsilon, \overline{\mathsf{st}})$. Since M does not have stubs, the bisimulation lemma is applicable: if one of the configurations can make a step, so does the other, and the step leads to a pair of related configurations such that the lemma can be applied again. If the program terminates and does not err, both sides arrive to final configurations where environments are guaranteed to coincide by O-Stack. □

5 EMPIRICAL STUDY

Anecdotal evidence suggests that type stability is discussed in the Julia community, but does Julia code exhibit the properties of stability and groundedness in practice? And if so, are there any indicators correlated with instability and ungroundedness? To find out, we ran a dynamic analysis on a corpus of Julia packages. All the packages come from the main language registry and are readily available via Julia's package manager; registered packages have to pass basic sanity checks and usually have a test suite.

The main questions of this empirical study are:

(1) How uniformly are type stability and groundedness spread over Julia packages? How much of a difference can users expect from different packages?
(2) Are package developers aware of type stability?
(3) Are there any predictors of stability and groundedness in the code and do they influence how type-stable code looks?

5.1 Methodology

We take as our main corpus the 1000 most starred (using GitHub stars) packages from the Julia package registry; as of the beginning of 2021, the registry contained about 5.5K packages. The main corpus is used for an automated, high-level, aggregate analysis. We also take the 10 most starred packages from the corpus to perform finer-grained analysis and manual inspection. Out of the 1000 packages in the corpus, tests suits of only 760 succeeded on Julia 1.5.4, so these 760 comprise our final corpus. The reasons of failures are diverse, spanning from missing dependencies to the absence of tests, to timeout.

For every package of interest, the dynamic analysis runs the package test suite, analyzes compiled method instances, and records information relevant to type stability and groundedness. Namely, once a test suite runs to completion, we query Julia's virtual machine for the current set of available method instances, which represent instances compiled during the tests' execution. To avoid bias towards the standard library, we remove instances of methods defined in standard modules, such as `Base`, `Core`, etc., which typically leaves us with several hundreds to several thousands of instances. For these remaining instances, we analyze their type stability and groundedness. As type information is not directly available for compiled, optimized code, we retrieve the original method of an instance and run Julia's type inferencer to obtain each register's type, similarly to the Jules model. In rare cases, type inference may fail; on our corpus, this almost never happened, with at most 5 failures per package. With the inference results at hand, we check the concreteness of the register typing and record a yes/no answer for both stability and groundedness. In addition to that, several metrics are recorded for each method: method size, the number of gotos and returns in the body, whether the method has varargs or `@nospecialize` arguments[6], and how polymorphic the method is, i.e. how many instances were compiled for it. This information is then used to identify possible correlations between the metrics and stability/groundedness.

To get a better understanding of type stability and performance, we employ several additional tools to analyze the 10 packages. For example, we look at their documentation, especially at the stated goals and domain of a package, and check the Git history to see if and how type stability is mentioned in the commits.

5.2 Package-Level Analysis

The aggregate results of the dynamic analysis for the 760 packages are shown in Table 1: 74% of method instances in a package are stable and 57% grounded, on average; median values are close to

[6] `@nospecialize` tells the compiler to *not* specialize for that argument and leave it abstract.

Table 1. Aggregate statistics for stability and groundedness

	Stable	Grounded
Mean	74%	57%
Median	80%	57%
Std. Dev.	22%	24%

the means. The standard deviation is noticeable, so even on small samples of packages, we expect to see packages with large deflections from the means.

A more detailed analysis of the 10 most starred packages, in alphabetical order, is shown in Table 2. A majority of these packages have stability numbers very close to the averages shown above, with the exception of Knet, which has only 16% of stable and 8% of grounded instances.

Table 2. Stability, groundedness, and polymorphism in 10 popular packages

Package	Methods	Instances	Inst/Meth	Varargs	Stable	Grounded
DifferentialEquations	1355	7381	5.4	3%	70%	44%
Flux	381	4288	11.3	13%	76%	70%
Gadfly	1100	4717	4.3	10%	81%	58%
Gen	973	2605	2.7	2%	64%	43%
Genie	532	1401	2.6	12%	93%	78%
IJulia	39	136	3.5	8%	84%	60%
JuMP	2377	36406	15.3	7%	83%	63%
Knet	594	9013	15.2	7%	16%	8%
Plots	1167	5377	4.6	8%	74%	58%
Pluto	727	2337	3.2	4%	80%	66%

The Knet package is a type stability outlier. A quick search over project's documentation and history shows that the only kind of stability ever mentioned is numerical stability; furthermore, the word "performance" is mostly used to reference the performance of neural networks or CUDA-related utilities. Indeed, the package primarily serves as a communication layer for a GPU; most computations are done by calling the CUDA API for the purpose of building deep neural networks. Thus, in this specific domain, type stability of Julia code appears to be irrelevant.

On the other side of the stability spectrum is the 93% stable (78% grounded) Genie package, which provides a web application framework. Inspecting the package, we can confirm that its developers were aware of type stability and intentional about performance. For example, Genie's Git history contains several commits mentioning (improved) "type stability." The project README file states that the authors build upon

"Julia's strengths (high-level, high-performance, dynamic, JIT compiled)."

Furthermore, the tutorial claims:

"Genie's goals: unparalleled developer productivity, excellent run-time performance."

Type stability (non-)correlates. One parameter that we conjectured may correlate with stability is the average number of method instances per method (Inst/Meth column of Table 2), as it expresses the amount of polymorphism discovered in a package. Most of the packages compile just 2–4 instances per method on average, but Flux, JuMP, and Knet have this metric 5–6 times higher, with JuMP and Knet exploiting polymorphism the most, at 15.3 and 15.2 instances per method,

respectively. The latter may be related to the very low type stability index of Knet. However, the other two packages are more stable than the overall average. Analyzing JuMP and Flux further, we order their methods by the number of instances. In JuMP, the top 10% of most instantiated methods are 5% less stable and grounded than the package average, whereas in Flux, the top 10% have about the same stability characteristics as on average. Overall, we cannot conclude that method polymorphism is related to type stability.

Another dimension of polymorphism is the variable number of arguments in a method (Varargs column of Table 2). We looked into three packages with a higher than average (9%) number of varargs methods in the 10 packages: Flux, Gadfly and Genie. Relative to the total number of methods, Flux has the most varargs methods—13%—and those methods are only 55% stable and 44% grounded, which is a significant drop of 21% and 26% below this package's averages. However, the other two packages have higher-than-package-average stability rates, 82% (Gadfly) and 99% (Genie), with groundedness being high in Genie, 93%, and low in Gadfly, 38%. Again, no general conclusion about the relationship between varargs methods and their stability can be made.

5.3 Method-Level Analysis

In this section, we inspect stability of individual methods in its possible relationship with other code properties like size, control flow (number of goto and return statements), and polymorphism (number of compiled instances and varargs). Our analysis consists of two steps: first, we plot histograms showing the number of methods with particular values of properties, and second, we manually sample some of the methods with more extreme characteristics.

5.3.1 Graphical Analysis. We use two-dimensional histograms like those presented in Fig. 11 to discover possible relationships between stability of code and its other properties. The vertical axis measures stability (on the left diagram) or groundedness (on the right): 1 means that all recorded instances of a method are stable/grounded, and 0 means that none of them are. The horizontal axis measures the property of interest; in the case of Fig. 11, it is method size (actual numbers are not important here: they are computed from Julia's internal representation of source code). The color of an area reflects how many methods have characteristics corresponding to that area's position on the diagram; e.g. in Fig. 11, the lonely yellow areas indicate that there are about 500 (400) small methods that are stable (grounded).

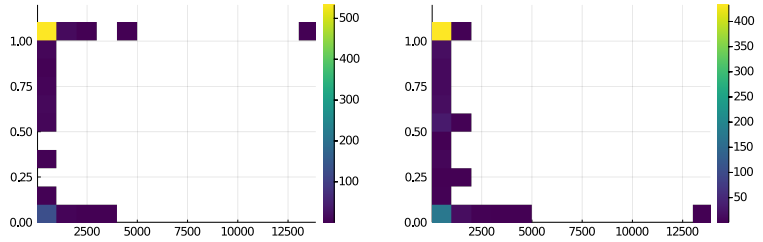

Fig. 11. Stability (left, OY axis) and groundedness (right, OY) by method size (OX) in Pluto

We generate graphs for all of the 10 packages listed in Table 2, for all combinations of the properties of interest; the graphs are available in the extended version [Pelenitsyn et al. 2021b]. Most of the graphs look very similar to the ones from Fig. 11, which depicts Pluto—a package for creating Jupyter-like notebooks in Julia. In the following paragraphs, we discuss features of these graphs and highlight the discrepancies.

The first distinctive feature of the graphs is the hot area in the top-left corner: most of the 10 packages employ many small, stable/grounded methods; the bottom-left corner is usually the second-most hot, so a significant number of small methods are unstable/ungrounded. For the Knet package, these two corners are reversed; for DifferentialEquations, they are reversed only on the groundedness plot. Both of these facts are not surprising after seeing Table 2, but having a visual tool to discover such facts may be useful for package developers.

The second distinctive feature of these graphs is the behavior of large, ungrounded methods (bottom-right part of the right-hand-side graph). The "tail" of large methods on the groundedness graphs almost always lies below the 1-level; furthermore, larger methods tend to be less grounded. However, if we switch from groundedness to stability plots, a large portion of the tail jumps to the 1-level. This means larger methods are unlikely to be grounded (as expected, because of the growing number of registers), but they still can be stable and thus efficiently used by other methods. Pluto provides a good example of such a method: its `explore!` method of size 13003 (right-most rectangle on Fig. 11, 330 lines in the source code) analyzes Julia syntax trees for scope information with a massive `if/else if/..` statement. This method has a very low chance of being grounded, and it was not grounded on the runs we analyzed. However, the method has a concrete return type annotation, so Julia (and the programmer) can easily see that it is stable.

In the case of the number of gotos and returns, the plots are largely similar to the ones for method size, but they highlight one more package with low groundedness. Namely, the Gen package (aimed at probabilistic inference [Cusumano-Towner et al. 2019]) has the hottest area in the bottom-left corner, contrary to the first general property we identified for the size-based plots. Recall (Tables 1 and 2) that Gen's groundedness is 14% less than the average on the whole corpus of 760 packages.

5.3.2 Manual Inspection. To better understand the space of stable methods, we performed a qualitative analysis of a sample of stable methods that have either large sizes or many instances.

Many large methods have one common feature: they often have a return type ascription on the method header of the form:

```
function f(...) :: Int
  ...
end
```

These ascriptions are a relatively new tool in Julia, and they are used only occasionally, in our experience. An ascription makes the Julia compiler insert implicit conversions on all return paths of the method. Conversions are user extendable: if the user defines type `A`, they can also add methods of a special function `convert` for conversion to `A`. This function will be called when the compiler expects `A` but infers a different type, for example, if the method returns `B`. If the method returns `A`, however, then `convert` is a no-op.

Type ascriptions may be added simply as documentation, but they can also be used to turn type instability into a run-time error: if the ascribed type is concrete and a necessary conversion is not available, the method will fail. This provides a useful, if unexpected, way to assure that a large method never becomes unstable.

While about 85% of type-stable methods in the top 10 packages are uninteresting in that they always return the same type, sampling the rest illuminates a clear pattern: the methods resemble what we are used to see in statically typed languages with parametric polymorphism. Below is a list of categories that we identify in this family.

- Various forms of the identity function—a surprisingly popular function that packages keep reinventing. In an impure language, such as Julia, an identity function can produce various

side effects. For example, the Genie package adds a caching effect to its variant of the identity function:

```
# Define the secret token used in the app for encryption and salting.
function secret_token!(value::AbstractString=Generator.secret_token())
  SECRET_TOKEN[] = value
  return value
end
```

- Container manipulations for various kinds of containers, such as arrays, trees, or tuples. For instance, the latter is exemplified by the following function from Flux, which maps a tuple of functions by applying them to the given argument:

```
function extraChain(fs::Tuple, x)
  res = first(fs)(x)
  return (res, extraChain(Base.tail(fs), res)...)
end
extraChain(::Tuple{}, x) = ()
```

- Smart constructors for user-defined polymorphic structures. For example, the following convenience function from JuMP creates an instance of the `VectorConstraint` structure with three fields, each of which is polymorphic:

```
function build_constraint(_error::Function, Q::Symmetric{V,M}, ::PSDCone)
        where {V<:AbstractJuMPScalar,M<:AbstractMatrix{V}}
    n = LinearAlgebra.checksquare(Q)
    shape = SymmetricMatrixShape(n)
    return VectorConstraint(
        vectorize(Q, shape),
        MOI.PositiveSemidefiniteConeTriangle(n),
        shape)
end
```

- Type computations—an unusually wide category for a dynamically typed language. Thus, for instance, the Gen package defines a type that represents generative functions in probabilistic programming, and a function that extracts the return and argument types:

```
# Abstract type for a generative function with return value type T and trace type U.
abstract type GenerativeFunction{T,U <: Trace} end
get_return_type(::GenerativeFunction{T,U}) where {T,U} = T
get_trace_type(::GenerativeFunction{T,U}) where {T,U} = U
```

5.4 Takeaways

Our analysis shows that a Julia user can expect mostly stable (74%) and somewhat grounded (57%) code in widely used Julia packages. If the authors are intentional about performance and stability, as demonstrated by the Genie package, those numbers can be much higher. Although our sample of packages is too small to draw strong conclusions, we suggest that several factors can be used by a Julia programmer to pinpoint potential sources of instability in their package. For example, in some cases, varargs methods might indicate instability. Large methods, especially ones with heavy control flow, tend to not be type grounded but often are stable; in particular, if they always return the same concrete type. Finally, although highly polymorphic methods are neither stable nor unstable in general, code written in the style of parametric polymorphism often suggests type stability.

Our dynamic analysis and visualization code is written in Julia (and some bash code), and relies on the vanilla Julia implementation. Thus, it can be employed by package developers to study type instability in their code, as well as check for regressions.

6 RELATED WORK

Type stability and groundedness are the consequences of Julia's compilation strategy put into practice. The approach Julia takes is new and simpler than other approaches to efficient compilation of dynamic code.

Attempts to efficiently compile dynamically dispatched languages go back nearly as far as dynamically dispatched languages themselves. Atkinson [1986] used a combination of run-time-checked user-provided types and a simple type inference algorithm to inline methods. Chambers and Ungar [1989] pioneered the just-in-time model of compilation in which methods are specialized based on run-time information. Hölzle and Ungar [1994] followed up with method inlining optimization based on recently observed types at the call site. Rigo [2004] specialized methods on invocation based on their arguments, but this was limited to integers. Similarly, Cannon [2005] developed a type-inferring just-in-time compiler for Python, but it was limited by the precision of type inference. Logozzo and Venter [2010] extended this approach with a more sophisticated abstract interpretation-based inference system for JavaScript.

At the same time, trace-based compilers approached the problem from another angle [Chang et al. 2007]. Instead of inferring from method calls, these compilers had exact type information for variables in straight-line fragments of the program called traces. Gal et al. [2009] describes a trace-based compiler for JavaScript that avoids some pitfalls of type stability, as traces can cross method boundaries. However, it is more difficult to fix a program when tracing does not work well, for the boundaries of traces are not apparent to the programmer.

Few of these approaches to compilation have been formalized. Guo and Palsberg [2011] described the core of a trace-based compiler with two optimizations, variable folding and dead branch/store elimination. Myreen [2010] formalized self-modifying code for x86. Finally, Flückiger et al. [2018] formally described speculation and deoptimization and proved correctness of some optimizations; Barriere et al. [2021] extended and mechanized these results.

The Julia compiler uses standard techniques, but differs considerably in how it applies them. Many production just-in-time compilers rely on static type information when it is available, as well as a combination of profiling and speculation [Duboscq et al. 2013; Würthinger et al. 2012]. Speculation allows these compilers to perform virtual dispatch more efficiently [Flückiger et al. 2020]. Profiling allows for tuning optimizations to a specific workload [Ottoni 2018; Xu et al. 2009], eliminating overheads not required for cases observed during execution. Julia, on the other hand, performs optimization only once per method instance. This presents both advantages and issues. For one, Julia's performance is more predictable than that of other compilers, as the warmup is simple [Barrett et al. 2017]. Overall, Julia is able to achieve high performance with a simple compiler.

7 CONCLUSION

Julia's performance is impressive, as it improves on that of a statically typed language such as Java, whose compiler has been developed for two decades by an army of engineers. This is achieved by a combination of careful language design, simple but powerful compiler optimizations, and disciplined use of abstractions by programmers.

In this paper, we formally define the notions of type stability and groundedness, which guide programmers towards code idioms the compiler can optimize. To this end, we model Julia's intermediate representation with an abstract machine called Jules, prove the correctness of its JIT compiler, and show that type groundedness is the property that enables the compiler to devirtualize

method calls. The weaker notion of type stability is still useful as it allows callers of a function to be grounded. This relationship between groundedness and stability explains the discrepancy between the definition of stability and what programmers do in practice, namely, inspecting the compiler's output to look for variables that have abstract types. Our corpus analysis of Julia packages shows that more than half of compiled methods are grounded, and over two-thirds are stable. This suggests that developers follow type-related performance guidelines.

Although our analysis suggests high presence of type-stable code, some Julia packages, even among more popular ones, have significant unstable portions. This may indicate an oversight from the package authors, albeit an understandable one: the tools for detecting unstable code are limited.[7] There is a future work opportunity in developing a user-facing static type system that would soundly determine whether a method is type stable or not. This could provide the benefits of a traditional (gradual) type system, and additionally, allow programmers to ensure type stability. A less demanding approach to facilitating stability would be to add primitives for reifying the programmer's intent. For example, the user could write `assert(is_type_stable())` to indicate that the method does not tolerate inefficient compilation.

Although the specific optimizations enabled by grounded code may not be as important for other languages, they can benefit from the lessons learned from Julia's performance model: namely, that the interface exposed by a compiler can be just as important as the cleverness of the compiler itself. Performance is not an isolated property: it is the result of a dialogue between the compiler and the programmers that use it.

DATA AVAILABILITY STATEMENT

The paper is accompanied by the artifact [Pelenitsyn et al. 2021a] reproducing the results of Sec. 5. In particular, the artifact contains the list of 1000 Julia packages analyzed (with the exact package versions), as well as Bash and Julia scripts performing stability analysis and generating Tables 1 and 2, and Fig. 11.

ACKNOWLEDGMENTS

We thank Ming-Ho Yee and the anonymous reviewers for their insightful comments and suggestions to improve this paper.

This work was supported by Office of Naval Research (ONR) award 503353, the National Science Foundation awards 1759736, 1925644, 1618732, CCF-1909143 and CCF-1908389 the Czech Ministry of Education from the Czech Operational Programme Research, Development, and Education, under grant agreement No. CZ.02.1.01/0.0/0.0/15_003/0000421, and the European Research Council under the European Union's Horizon 2020 research and innovation programme, under grant agreement No. 695412.

REFERENCES

Gerald Aigner and Urs Hölzle. 1996. Eliminating Virtual Function Calls in C++ Programs. In *European Conference on Object-Oriented Programming (ECOOP)*. https://doi.org/10.1.1.7.7766

Robert G. Atkinson. 1986. Hurricane: An Optimizing Compiler for Smalltalk. In *Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*. https://doi.org/10.1145/960112.28712

Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount, and Laurence Tratt. 2017. Virtual Machine Warmup Blows Hot and Cold. *Proc. ACM Program. Lang.* 1, OOPSLA (2017). https://doi.org/10.1145/3133876

Aurele Barriere, Olivier Flückiger, Sandrine Blazy, David Pichardie, and Jan Vitek. 2021. Formally Verified Speculation and Deoptimization in a JIT Compiler. *Proc. ACM Program. Lang.* 5, POPL (2021). https://doi.org/10.1145/3434327

[7] Besides manually calling `@code_warntype`, one can use JET.jl package in "performance linting" mode. JET relies on Julia's type inference; its primary goal is to report likely dynamic-dispatch errors, and it does not provide sound analysis.

Julia Belyakova, Benjamin Chung, Jack Gelinas, Jameson Nash, Ross Tate, and Jan Vitek. 2020. World Age in Julia: Optimizing Method Dispatch in the Presence of Eval. *Proc. ACM Program. Lang.* 4, OOPSLA (2020). https://doi.org/10.1145/3428275

Jeff Bezanson, Jiahao Chen, Ben Chung, Stefan Karpinski, Viral B. Shah, Jan Vitek, and Lionel Zoubritzky. 2018. Julia: Dynamism and Performance Reconciled by Design. *Proc. ACM Program. Lang.* 2, OOPSLA (2018). https://doi.org/10.1145/3276490

Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. 2017. Julia: A Fresh Approach to Numerical Computing. *SIAM Rev.* 59, 1 (2017). https://doi.org/10.1137/141000671

Brett Cannon. 2005. *Localized Type Inference of Atomic Types in Python.* Master's thesis. California Polytechnic State University.

Craig Chambers and David Ungar. 1989. Customization: Optimizing Compiler Technology for SELF, a Dynamically-Typed Object-Oriented Programming Language. In *Proceedings of the ACM Programming Language Design and Implementation Conference (PLDI).* 146–160. https://doi.org/10.1145/73141.74831

Mason Chang, Michael Bebenita, Alexander Yermolovich, Andreas Gal, and Michael Franz. 2007. *Efficient just-in-time execution of dynamically typed languages via code specialization using precise runtime type inference.* Technical Report. Donald Bren School of Information and Computer Science, University of California, Irvine, 2007.

Marco F. Cusumano-Towner, Feras A. Saad, Alexander K. Lew, and Vikash K. Mansinghka. 2019. Gen: A General-purpose Probabilistic Programming System with Programmable Inference. In *Proceedings of the ACM Programming Language Design and Implementation Conference (PLDI).* https://doi.org/10.1145/3314221.3314642

L. Peter Deutsch and Allan M. Schiffman. 1984. Efficient Implementation of the Smalltalk-80 System. In *ACM Symposium on Principles of Programming Languages (POPL).* https://doi.org/10.1145/800017.800542

Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. 2013. An Intermediate Representation for Speculative Optimizations in a Dynamic Compiler. In *Workshop on Virtual Machines and Language Implementations (VMIL).* https://doi.org/10.1145/2542142.2542143

Olivier Flückiger, Guido Chair, Ming-Ho Yee, Jan Jecmen, Jakob Hain, and Jan Vitek. 2020. Contextual Dispatch for Function Specialization. *Proc. ACM Program. Lang.* 4, OOPSLA (2020). https://doi.org/10.1145/3428288

Olivier Flückiger, Gabriel Scherer, Ming-Ho Yee, Aviral Goel, Amal Ahmed, and Jan Vitek. 2018. Correctness of speculative optimizations with dynamic deoptimization. *Proc. ACM Program. Lang.* 2, POPL (2018). https://doi.org/10.1145/3158137

Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. 2009. Trace-based just-in-time type specialization for dynamic languages. In *Programming Language Design and Implementation (PLDI).* https://doi.org/10.1145/1543135.1542528

Shu-yu Guo and Jens Palsberg. 2011. The Essence of Compiling with Traces. In *Symposium on Principles of Programming Languages (POPL).* https://doi.org/10.1145/1926385.1926450

U. Hölzle and D. Ungar. 1994. Optimizing dynamically-dispatched calls with run-time type feedback. In *Programming Language Design and Implementation (PLDI).* https://doi.org/10.1145/178243.178478

Francesco Logozzo and Herman Venter. 2010. RATA: Rapid Atomic Type Analysis by Abstract Interpretation – Application to JavaScript Optimization. In *Compiler Construction (CC).* https://doi.org/10.1007/978-3-642-11970-5_5

Magnus O. Myreen. 2010. Verified Just-in-Time Compiler on X86. In *Symposium on Principles of Programming Languages (POPL).* https://doi.org/10.1145/1706299.1706313

Guilherme Ottoni. 2018. HHVM JIT: A Profile-Guided, Region-Based Compiler for PHP and Hack. In *Programming Language Design and Implementation (PLDI).* https://doi.org/10.1145/3192366.3192374

Artem Pelenitsyn, Julia Belyakova, Benjamin Chung, Ross Tate, and Jan Vitek. 2021a. Type Stability in Julia: Avoiding Performance Pathologies in JIT Compilation (Artifact). https://doi.org/10.5281/zenodo.5500548

Artem Pelenitsyn, Julia Belyakova, Benjamin Chung, Ross Tate, and Jan Vitek. 2021b. Type Stability in Julia: Avoiding Performance Pathologies in JIT Compilation (Extended Version). arXiv:2109.01950

Armin Rigo. 2004. Representation-Based Just-in-Time Specialization and the Psyco Prototype for Python. In *Partial Evaluation and Program Manipulation (PEPM).* https://doi.org/10.1145/1014007.1014010

Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. 2012. Self-Optimizing AST Interpreters. In *Dynamic Language Symposium (DLS).* https://doi.org/10.1145/2384577.2384587

Guoqing Xu, Matthew Arnold, Nick Mitchell, Atanas Rountev, and Gary Sevitsky. 2009. Go with the Flow: Profiling Copies to Find Runtime Bloat. In *Programming Language Design and Implementation (PLDI).* https://doi.org/10.1145/1542476.1542523

Francesco Zappa Nardelli, Julia Belyakova, Artem Pelenitsyn, Benjamin Chung, Jeff Bezanson, and Jan Vitek. 2018. Julia Subtyping: A Rational Reconstruction. *Proc. ACM Program. Lang.* 2, OOPSLA (2018). https://doi.org/10.1145/3276483